An FPGA Based Architecture for Native Protocol Testing of Multi-Gbps Source-Synchronous Devices

A Dissertation
Presented to
The Academic Faculty

By

Carl E. Gray

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Electrical and Computer Engineering

Georgia Institute of Technology

August 2012

An FPGA Based Architecture for Native Protocol Testing of Multi-Gbps Source-Synchronous Devices

Approved by:

Dr. David C. Keezer, Advisor
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Linda M. Wills
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Linda S. Milor
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Keren Bergman
Department of Electrical Engineering
Columbia University

Dr. Kevin P. Martin
Microelectronics Research Center
Georgia Institute of Technology

Date Approved: June 29, 2012

AKNOWLEDGEMENTS

I would like to express my thanks and gratitude to my advisor Dr. David Keezer for his significant contributions in time and effort to my development as a student and a researcher. Without his continued patience and guidance I would not have made it to this point.

I would also like to thank Dr. Linda Milor and Dr. Kevin Martin for serving on the proposal committee and as committee members for my Ph.D. dissertation and for their constructive comments, suggestions, and support. I would especially like to thank Dr. Linda Wills for serving as a committee member for my Ph.D. defense after the untimely passing of her husband Dr. Scott Wills who served as the chairman of my proposal committee and was a co-contributor to the sponsoring research effort that propelled much of this work. Without his energy and enthusiasm, as a lecturer, researcher, and mentor, I may have lost my own enthusiasm and momentum.

A special thanks goes to Dr. Odile Liboiron-Ladouceur, Howard Wang, and Dr. Keren Bergman, who also gratefully served on my defense committee, for making me feel like a valued guest on my collaboration visits to the Lightwave Research Laboratory at Columbia University. Without your novel systems, research, and cooperation the very foundation and genesis of this research would have never existed.

Finally, I would like to thank all of my family and friends who's support and encouragement throughout my time at Georgia Tech has been invaluable and cannot be sufficiently put into words. Thank you.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS AND ABBREVIATIONS

AC	Alternating Current
ATE	Automatic Test Equipment
BIST	Built in Self Test
CDR	Clock and Data Recovery
CPU	Central Processign Unit
CUT	Circuit Under Test
DAC	Digital-to-Analog Converter
DC	Direct Current
DDR	Double Data Rate
DFT	Design For Test
DLC	Digital Logic Core
DLL	Delay Locked Loop
DTC	Digital Test Core
DUT	Device Under Test
DV	Data Vortex
DWDM	Dense Wavelength Division Multiplexing
E/O	Electrical to Optical
FPGA	Field Programmable Gate Array
GVD	Group Velocity Dispersion
I/O	Input and Output
I2C	Inter-Integrated Circuit
I2S	Inter-IC Sound
IP	Intellectual Property
JTAG	Joint Test Action Group
LFRS	Linear Feedback Shift Register
$LiNbO_3$	Lithium Niobate
LUT	Look Up Table
NRZ	Non-return to Zero

O/E	Optical to Electrical
OPS	Optical Packet Switched
PA-ATE	Protocol Aware ATE
PCB	Printed Circuit Board
PCIE	PCI Express
PECL	Positive Emitter-coupled Logic
PLL	Phase Locked Loop
R1	Return to One
RC	Return to Complement
ROM	Read Only Memory
RZ	Return to Zero
SiGe	Silicon Germanium
SMA	SubMiniature version A
SMP	SubMiniature Push-On
SOA	Semiconductor Optical Amplifier
SOC	System on a Chip
SPI	Serial Peripheral Interface
TAP	Test Access Port
TCK	Test Clock
TDI	Test Data In
TDO	Test Data Out
TIA	Transimpedance Amplifier
TMS	Test Mode Select
TRST	Test Reset
USB	Universal Serial Bus
VHDL	VHSIC Hardware Description Language
WLP	Wafer Level Package

SUMMARY

This thesis presents methods for developing FPGA-based test solutions that solve the challenges of evaluating source-synchronous and protocol-laden systems and devices at multi-gigabit per second signaling rates. These interfaces are becoming more prevalent in emerging designs and are difficult to test using traditional automated test equipment (ATE) and test instrumentation which were designed for testing designs utilizing synchronous and deterministic signaling. The main motivation of this research was to develop solutions that address these challenges.

The methods shown in this thesis are used to design a test architecture consisting of custom hardware components, reprogrammable digital logic for hardware integration, and a software interface for external data transport and configuration. The hardware components consist of a multi-GHz field programmable gate array (FPGA) based interface board providing processing, control, and data capabilities to the system and enhanced by one or more application modules which can be tailored for specific test functionality compatible with source-synchronous and protocol interfaces. Software controls from a host computer provide high and low level access to the internal tester data and configuration memory space.

The architecture described in this thesis is demonstrated through a specific test solution for a high-speed optical packet switched network called the Data Vortex. Reprogrammable firmware and software controls allow for a high degree of adaptability and application options. The modularized implementation of the hardware elements introduces additional adaptability and future upgradability, capable of incorporating new materials and design techniques for the test platform and application modules.

CHAPTER 1

INTRODUCTION

The objective of this research is to develop an adaptable, field-programmable gate-array (FPGA) based architecture for native protocol testing of multi-Gbps source-synchronous systems and devices. Existing automated test equipment (ATE) systems and benchtop instrumentation suites are incapable of, or ill suited to, the task of evaluating this category of systems due to the high-bandwidth, wide channel, non-deterministic, and protocol laden interactions that these system interfaces employ. A design architecture, utilizing reprogrammable modern FPGA technology and application specific extension modules, is presented in this research intended to allow for the creation of test designs capable of being rapidly adapted for native protocol testing of current and emerging designs that are otherwise prohibitively difficult to test with existing systems and methodologies. This research presents a physical system implemented using this test architecture, and the performance and capability of this system is demonstrated through the test and evaluation of the Data Vortex optical packet switching network.

Testing of devices and systems is an integral part of the design and manufacturing process. The exact methodology and systems utilized vary depending on the specific stage of the process and the information to be gathered, but the end intent remains the same: to verify that the device or system is functioning as designed and within the desired operational parameters. The development of devices and systems is outpacing the general ability to test the resulting designs in native operational modes and at full design speed. While the performance of these devices has consistently increased in clock and data rates, the respective performance of automated test equipment has tended to be incremental and lag the leading edge components [1]. This prohibits the ability to test devices at native operating speeds and in "mission mode" which is becoming a critical

concern as the failure modes become more subtle and timing related [2]. Benchtop instrumentation is often capable of performance exceeding that of ATE but these units are also expensive, channel limited, and function specific.

Furthermore, the recent trend of incorporating high-speed source-synchronous signaling techniques, for which the conventional ATE test methodology was not designed to address, into newer designs further complicates the testing challenges using these test systems. The conventional ATE architecture has been designed for the testing of parallel I/O and data busses. These test systems use paired stimuli and response vectors that rely on a fixed timing relationship between the test system and the device under test (DUT) [3]. The non-deterministic signaling nature of source-synchronous data invalidates this relationship. Additionally, since the conventional ATE test infrastructure is dependent on pre-calculated signaling vectors and is not equipped for real-time processing and response generation to incoming signals, protocol based interfaces utilizing dynamic timing cannot be natively supported.

This research presents a test architecture capable of interfacing and adapting to new and emerging systems incorporating some or all of these difficult test challenges which would otherwise be beyond the capability of current and conventional test techniques. This architecture is demonstrated through a physical implementation and the performance and capabilities of that system as tailored to a real test target.

The organization of this thesis is as follows. In Chapter 2, a brief background of the principles of testing and the methodologies to achieve these goals is provided. A particular emphasis is placed on the evolution of the testing process and the reactionary measures enacted to adapt to the evolving test needs of devices and systems. Some of the latest design methodologies, such as FPGA based Digital Test Cores, ATE Extension, and Protocol Aware ATE, from which this research draws from or has been influenced by, are presented.

Chapter 3 introduces a novel optical packet switching network which is particularly challenging to test in a comprehensive yet economical fashion using conventional test systems. This network, the Data Vortex, has served both as the guiding motivation for the development of this test architecture and as the primary test bench for the proof and validation of the physical hardware implementations created under this research. The design of this network, its physical implementation, and some testing challenges associated with it are presented to explain and justify the some of the design parameters included in the performance requirements of this test system.

However, this tester architecture needs to not be specific to the Data Vortex. To ensure that the design developed by this work is flexible in the present and the future, the hardware is implemented in two parts: a primary test development platform supporting the core FPGA of the design and removable application modules. The FPGA firmware programming can be altered as needed to generate, capture, analyze, and, if necessary, react to the system signals. The application modules can be added, removed, interchanged or upgraded altogether to accommodate a range of test capability for a particular system or adapt to a range of target systems. The design and development of this test platform, including early prototypes and the lessons learned, is detailed in Chapter 4. The development of modules for signal transmission and capture are discussed in Chapters 5 and 6 respectively.

The processing and analysis capabilities available with this test architecture are discussed in Chapter 7. Simple functional verification, comparable to an ATE system, is possible using an external computer, in conjunction with the FPGA, or completely within it. More complex capability, by virtue of the hardware synthesis capability of the FPGA, is also available in the form of protocol emulation, natively implementing the interface rather than simulating the behavior at a high level.

Chapter 8 presents some additional modules that have been constructed or designed to work with the test platform to provide additional signaling rates and

configuration options. Included are modules to increase the total number of supported channels or the signaling rate supported by individual channels. This added capability allows the system to continue pushing the boundary of test capability and support higher total data rates. Other utility modules, such as a memory expansion board or signal breakout board to support many more slow speed signals have been designed to expand the general support capabilities of the system for added utility or design variations not exclusively oriented at raw throughput. This allows for a different dimension of development and the support of more complex and feature rich test configurations. Finally, Chapter 9 presents the conclusions, contributions, and potential future applications of this work.

CHAPTER 2

BACKGROUND

The testing of a device or system consists of exerting a stimuli and analyzing the result to evaluate the validity of the system response. The nature of the stimuli and the fashion in which it is exerted depends on the system complexity and the level of detail required by the test [4]. Another test concern is the nature of the device to be tested, as the testing requirements of a production device are significantly different from that of a new design. A new design requires extensive characterization involving comprehensive analysis of AC and DC measurements, design debugging, and verification. These tests allow for the discovery of the exact limits of the device operating range and are essential to the iterative design process. A production device or system can be subjected to a less comprehensive test that has been designed to have a high coverage of the possible faults discovered through the characterization process or from simulation models [5].

ATE hardware is heavily used in both the design verification process of new designs and systems as well as for production testing. Available from various manufacturers and in a wide range of configurations, the standard arrangement of an ATE system is a central unit that provides control, configuration, and function generators for the I/O interfaces. These interfaces consist of an array of reconfigurable pin electronics, numbering in the tens or hundreds per system operating at multi-Gbps data rates, and are designed to be adaptable to as wide a range of devices as possible. For example, a currently available system from Verigy, the V93000 SOC with features targeted at system-on-a-chip (SOC) product testing, can be populated with a total of 2048 pins operating between 800 Mbps and 3.6 Gbps and allowing for edge placement resolution in the picosecond range. The fastest pins are capable of DC swings between -0.95V to 3.05 V at 1 mV of resolution and support a variety of termination schemes [6].

While ATE systems are highly flexible, capable of adapting to test a wide range of devices and systems as well as provide a very high level of cost efficiency as measured per I/O, there are some applications where such systems cannot be or are impractical to be employed. For instance, there are many styles of tests intended to determine if the device has been manufactured correctly and attempt to detect these faults below the operational frequencies for the design being tested. However, there is a growing variety of faults and failures that can only be detected with the device operating at native operating frequencies [7]. While test solutions can be acquired at state-of-the-art data rates at the time of purchase, the ever increasing clock rates of processors and systems can quickly outpace the performance capabilities of the tester. With costs on the order of a million dollars or more for the central control system and a few thousand per pin [8], it can be financially infeasible for a company to constantly replace their ATE inventory to maintain a performance edge over the devices to be tested.

Some of these tests can be achieved using higher performance benchtop instrumentation such as bit error rate (BER) testers operating at extended frequencies up to 40 Gbps [9]. While such instruments can achieve an improvement over ATE with regards to performance they can be less time efficient, less cost efficient per channel, have lower channel density, and be less adaptable outside the specific intended application of the test instrument. There have also been some efforts to extend the usable range of ATE through a variety of modular systems and add-ons. These methods include approaches to multiplex and demultiplex collections of ATE channels to support interfacing with higher speed devices [10], to assist implementing protocols not natively supported by the ATE [11], or add additional circuits designed to function autonomously or as a slave to the host ATE system [12]. These approaches are designed to permit an ATE system to test a device the system would otherwise be incapable of testing, but these solutions add an additional expense and still require the expensive core infrastructure of the underlying ATE.

There is also an increase in the use and complexity of protocol laden interfaces at the system boundary which reduces the efficiency of conventional test techniques and complicates the usage of systems that are not designed to natively operate with these interfaces. If not adapted through one of the above solutions, conventional ATE can only interact with these target devices and systems at the lowest physical level, requiring higher level synthesis of signal and timing vectors to execute a test at the system level. These test systems operate on pre-calculated test vectors which are a seemingly arbitrary "sea of bits" that is not interpretable or editable at a high level of abstraction resulting in a difficult and time consuming process to alter the test pattern [13]. A Protocol Aware (PA) ATE system, defined as having "the ability for the ATE pin or pin group to natively emulate real time chip I/O at the protocol level" [14], would open up the ability to interact with the device under test on a much higher conceptual and architectural level that allows for a more fluid and dynamic test process. Some simple systems, supporting single channel serial interfaces, are just beginning to emerge with commercial availability [15].

The test architecture proposed by this work is designed to address some of these testing challenges by creating a miniature test system capable of interfacing to a high-speed protocol enabled device or system at its native operating mode and is able to exercise that interface at a high level of abstraction. The FPGA pin I/Os replicate in part the pins and pin groupings that can be found in conventional ATE with a system size and cost which are more in line with bench instrumentation. These pin I/Os are also augmented with external application modules and the programmable processing capability of the FPGA internal logic enabling the test system to implement the necessary protocols required for interacting with a target device or system using its native interface. While the overall resulting system may be more limited than the capabilities of ATE or high-performance bench instrument, many of the essential features such as adjustable signaling format, variable signal levels, and timing adjustment capability still allows for

the signal channels to interface to a wide variety of destination and source hardware modules. This flexibility can be used to further augment the signaling capabilities beyond that directly achievable by the FPGA alone.

2.1 Testing Principles

The basic goal of testing is to evaluate if a device or system is usable as designed. This can be achieved through evaluation to determine if the device is constructed properly, assuming the design has been previously validated, or to determine if the device functions properly within a desired set of performance parameters. As designs get larger, more complex, and operate at higher-speeds, the testing challenges get more complex. To meet these challenges, test techniques have continued to evolve, incorporating advanced design elements to improve the testability of a device or to even allow the device to assist in its own evaluation. Some of these testing principles are presented below.

2.1.1 Structural and Functional Test

Structural testing, also known as white box testing, uses knowledge of the actual internal implementation of a system [16]. Through this knowledge, test patterns can be created that attempt to energize and sensitize as much of the logic as possible in an attempt to find structural, manufacturing, or logical faults. Visually and conceptually, these test patterns are meaningless strings of ones and zeros. They are designed as Boolean combinations of input sequences to achieve as high a test coverage as possible while taking the smallest amount of time to apply and complete. These test sequences or vectors can be calculated by hand for a simple circuit or generated programmatically from a larger and more complex design utilizing knowledge of the physical design. Multiple test variants can be stored and configured to account for different forms of

analysis or as part of a diagnostic sequence to evaluate failures, but only one test sequence is active within the tester at a time.

Modern ATE systems utilize a per-pin architecture [17,18]. Each tester channel is comprised of a set of hardware features, specifically a test sequence memory, timing generator, and pin electronics, that is unique to that channel. Older system architectures shared resources, especially timing generators, which increased system complexity but reduced total costs. The per-pin architecture, while more expensive on a channel by channel basis, is linearly scalable to high channel counts and simplifies the test flow, shown in Figure 2.1, which results in improved system performance.

Tests are controlled from a central host computer, at the left of the figure, which is used to load tests and analyze results. The component elements of the test electronics, replicated once per pin within the ATE system, are shown in the center of the figure. Test patterns, either as stimuli for test input or captured response data for test outputs, are stored in a sequence memory which is accessible from the central computer. Timing generators are used to transform an input vector into a time domain representation of the desired signal or to capture the output at a specific timing value. The pin electronics are capable of generating a vast array of voltage levels, varying signal drive strength, and emulating termination schemes. The DUT or device under test, shown on the right, is the chip (also referred to as CUT or chip under test), device, board level elements, or a complete system being evaluated by the tester.

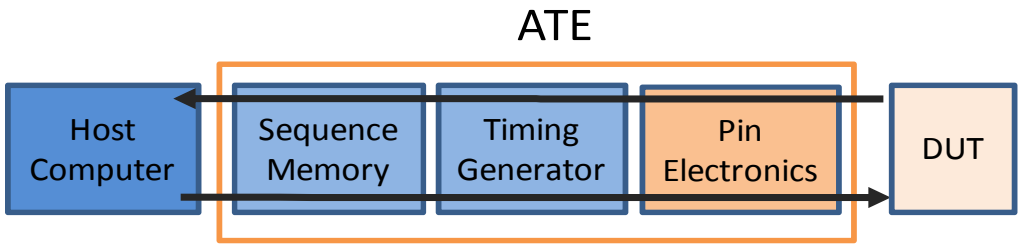


Figure 2.1 Basic test flow.

For a simple and non-stateful combinational logic circuit, it is possible to test the system using a static input and by observing the output at some point in time after the circuit output has stabilized. For more complex and sequential circuits, the timing and wave shape of the input signals and the response becomes as important if not more so than the values themselves [19]. Using these styles of signals, at-speed functional testing can be performed, evaluating the target system to see if it functions as desired under a standard input load. As with structural testing, these signal patterns can be constructed manually for a simple design or from a system simulation. Contrary to structural testing, these simulations do not require explicit understanding of the internal implementation of the DUT. A definition of the behavior of the output relative to the values applied to the inputs, referred to as a black box model since the specific internal implementation is unknown, can be utilized to construct a test meant to verify correct functioning of the system per those definitions [20].

In addition to the desired logical values for inputs and outputs, the design definition incorporates parameters such as voltage and current levels, setup and hold times, pulse width minimums and maximum, rise and fall times, etc. To satisfy these timing and format requirements, the signal channels for advanced testers incorporate timing generators capable of placing rising and falling edges of input signals and allowing the sampling point of output signals to be skewed in time. Combined with a variety of signal encodings, a wide range of signal formats can be created in the time domain.

One of the simplest signal encoding formats is non-return-to-zero (NRZ) in which the one and zero state are represented by a high and low voltage respectively. These voltage levels are held for the entire duration of the bit period. Another style of encoding holds the desired voltage for a percentage of the bit period and returns the value to a set point for the remainder of the cycle. These formats include return-to-zero (RZ) and return-to-one (R1). One last encoding format is return-to-complement (RC) in which the

desired value is encoded for part of the bit period and the complement is utilized for the remainder of the cycle.

Using combinations of these signal encodings, a range of time domain waveforms can be generated. For an NRZ signal, the placement of the rising edge can be designated to allow for time shifting the entire bit sequence for that channel, skewing it relative to all other channels unless those channels are also moved. In addition to this basic time skew capability, both the leading and trailing edges of the RZ, R1, and RC encoded signals can be adjusted. This allows for the creation of signals with a pulse width shorter than the overall tester bit period or to ensure compatibility with telecommunications applications which often utilize these style signal encodings. Return-to-zero formatting can also be utilized to create a clock pattern running at the same signaling rate as the tester. An NRZ encoded alternating clock pattern, 1010 repeating, would actually generate a clock at one half the tester operational rate. The HP 83000-F660 tester supports all of these encoding formats at a cycle time up to 660 MHz, edge timing accuracy of +/- 50 ps, and 10 ps edge placement resolution [21,22]. In combination, these parameters can be adjusted to create individual signal pulses down to 200 ps in width [23].

Some examples of these waveforms are shown in Figure 2.2. In this figure, the top most waveform is a non-return-to-zero (NRZ) 1010 sequence edge aligned with zero skew with respect to the tester cycle timing. Each signal cycle generated by the tester corresponds to a bit value that persists through the entire cycle. The second waveform is a return-to-zero (RZ) 1010 pattern with leading edges, which in this case correspond to logically rising transitions, located at the beginning of the tester cycle and trailing edges at the 50% midpoint. The third pattern is a return-to-one (R1) all zero pattern with leading edges delayed 25% from the beginning of the cycle and trailing edges delayed 75%. This results in very short zero pulses corresponding to the programmed sequence. The final waveform is a return-to-complement (RC) 0110 pattern with a non-skewed leading edge and 66% skewed trailing edge.

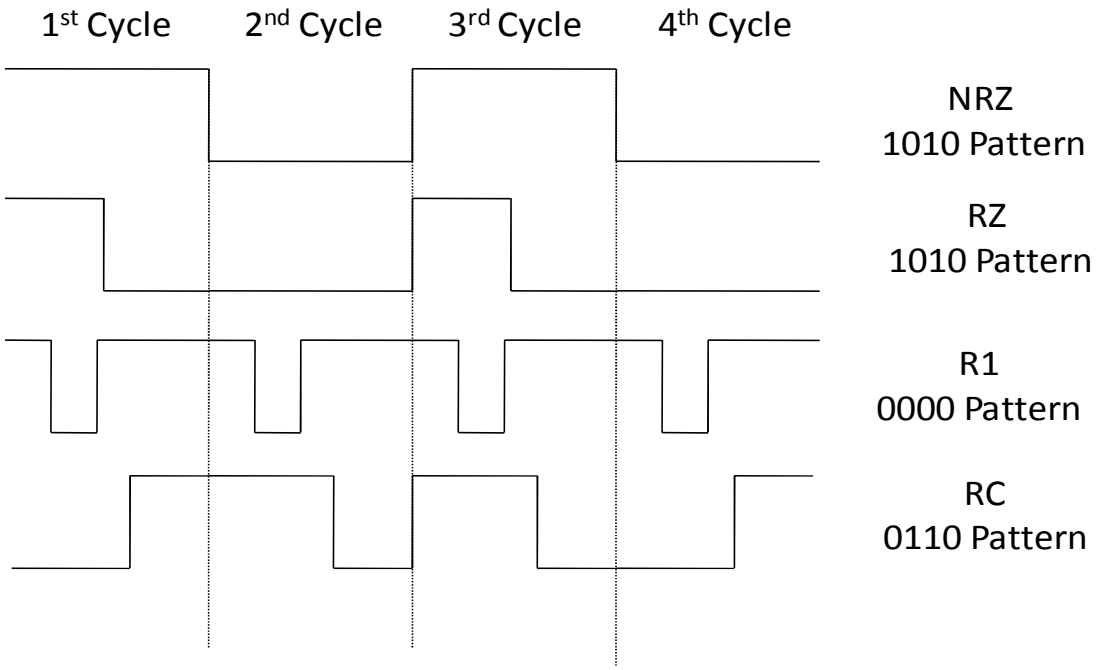


Figure 2.2 Time domain formatted waveforms.

The sampling time offset can also be varied to compensate for fixture skew, transmission propagation, and device latency. Output sampling timing can occur at a fixed placement as part of a test, manually varied to find operational parameters, or iterated across all possible combinations to generate a Shmoo plot [24]. Timing can be held constant to account for a specific desired timing relationship between signals or varied to find the operational range of the system.

2.1.2 Design-For-Test (DFT)

For older or smaller designs, it is possible to achieve acceptable coverage rates using test approaches only having access to the external pin interfaces. However, as manufacturing technology has continued to increase, device density has outstripped the ability to effectively test many designs. The ongoing trend, noted by Gordon Moore [25] and subsequently labeled as Moore's Law, is that the transistor density of devices has grown at an almost consistent rate, doubling per unit area roughly every 2 years. However, packaging has not kept similar pace. The logic density of a series of Xilinx

devices and the respective number of user configurable I/Os is shown in Table 1 [26]. The trends for other brands of FPGAs and other device technologies are very similar. As the disparity between internal logic and available pins for logic stimulus increases, effective test coverage diminishes.

Table 1 FPGA logic vs available user I/Os.

FPGA Model	Logic Cells	Effective Gates	Pin I/Os
XC2064	128	1024	58
XC3090	640	5120	144
XC4062	5472	43776	352
XC40200	16758	134064	448
XCV800	21168	169344	512
XC2V6000	67584	540672	1104
XC4VLX160	200448	1603548	960
XC6VLX550T	549888	4399104	1200
XC7V2000T	1954560	15636480	1200

In addition to density, design complexity continues to increase as well with corresponding increases to the required effort and costs to test these devices [27]. To improve test coverage and reduce tests costs, especially if those tests could be applied with low pin overhead, new approaches had to be created. One set of solutions are collectively referred to design-for-test or DFT. Design-for-test is the addition of logic or repurposing of existing logic in a design to enhance the testability of the underlying design. One of the most common implementations is the use of scan chains which are a series of registers configured as a large sequential shift register. Data can be shifted into or out of the device using a minimal number of device pins allowing access to data storage elements and logic deep within the device that would otherwise be inaccessible. Using this approach therefore adds an injection or observation point for new test data.

The registers utilized are either added to the system or, through the addition of small amounts of control logic, reused from existing logic elements in an auxiliary mode when the chain needs to be accessed. Data can be shifted into the design in a pure serial fashion or parallel loaded through special register cells around the perimeter of the device. Due to the location of these cells, these elements are referred to as Boundary Scan Cells.

The transport mechanism for this test chain solution is defined in IEEE Standard 1149.1 [28], often referred to as JTAG (which actually stands for Joint Test Action Group, the group that drafted the standard). A JTAG interface is a serial signaling system comprised of five signals. Due to the small number of pins required, this interface provides a very high amount of deep logic access with a minimum amount of overhead. The protocol is compatible with a single chip or can be chained to multiple devices, requiring only a single test point connection for a board level test solution. The internal components for a JTAG enabled device are shown in Figure 2.3 [29]. Boundary Scan cells are arrayed around the perimeter of the device, connected to both a serial shift loop and external pins for parallel load capability. The JTAG standard includes control elements such as the Test Access Port (TAP) controller and various data registers, such as the instruction, bypass, ID, and any other optional registers desired for a specific design.

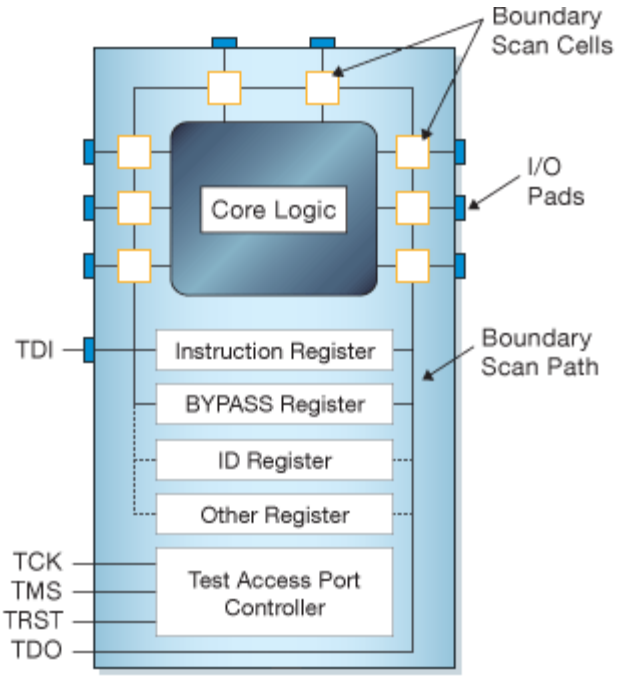


Figure 2.3 JTAG enabled device general schematic.

The signals required to implement the JTAG interface are Test Mode Select (TMS), Test Clock (TCK), Test Data In (TDI), and Test Data Out (TDO). These signals control the test mode settings, clock the system, and complete the serial data loop. Test Reset (TRST) is optional and may not be included in all implementations [30]. Individual elements can be accessed as shown in the figure or multiple chips or systems can daisy chained together, linking the TDI pin of the next device in the chain (or the connector interface if it's the first in the chain) to the TDO of the previous device (or the connector interface if it's the last in the chain). All elements receive a parallel and synchronously distributed copy of TCK and TMS.

Figure 2.4 shows how the four primary signals would be connected at the system level to connect multiple JTAG enabled devices to a single access connector [31]. TMS and TCK as control and clocking signals must be distributed to all destination devices in parallel. The data interface is implemented as a serial chain with TDI from the test signal header connected to the TDI pin of the first device. If there were only one device in the test chain, TDO from that device would be connected to the test signal header. If there

are multiple devices, the TDO of the first device is connected to the TDI pin of the next device in the chain and the process is repeated with the final device closing the loop to the signal header.

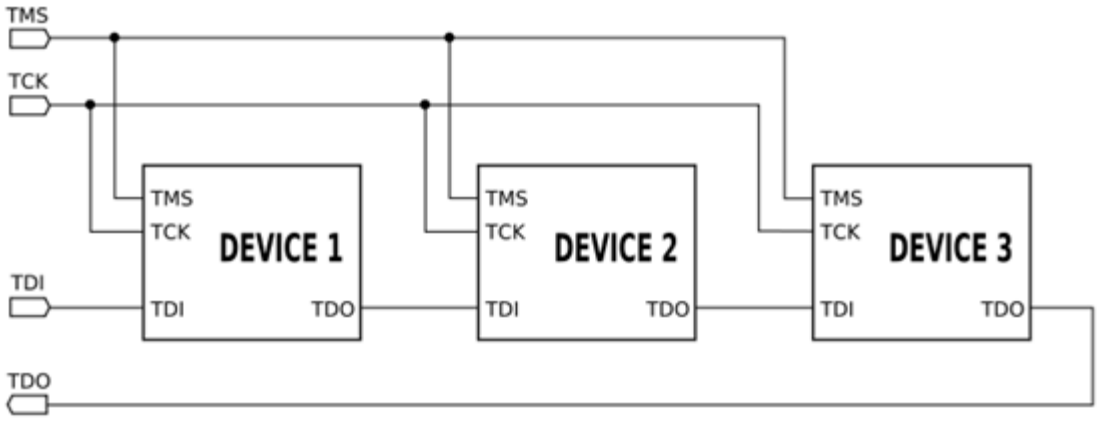


Figure 2.4 JTAG signal connections.

2.1.3 Built-in Self-Test (BIST)

Built-in self-test is a specific subset of DFT techniques through which test capability, including both the generation and application of tests, is accomplished using built-in hardware elements [32,33]. Much like a larger test solution, a basic BIST solution implementation is comprised of a pattern generator, a response analyzer, and a test controller. In combination, these elements allow a device to self evaluate the system status as fault-free or faulty [34]. Patterns can be statically stored in a read only memory (ROM) or generated dynamically through the use of a counter or linear feedback shift register (LFSR). Test output can be analyzed similarly, using stored responses or processed through a signature analyzer. Test controllers can be specific to individual test blocks or be common and reusable across a design tier, managing multiple test units simultaneously.

Scan chains can be valuable in BIST systems to setup tests or to observe the results and status of a completed test. Comprehensive results analysis can be performed,

comparable to what would be achieved with older style test implementations for manufacturing verification or characterization. Such a test could require that the device be placed into a special test condition that is contrary to the standard operational mode. A less complex and non-invasive test, including a simple pass/fail which can be part of the standard operational mode of the device, can be implemented to operate in parallel to detect real-time failures. More comprehensive tests, which may require the device or system to be taken offline, can be incorporated as an on-demand maintenance or diagnostic feature.

One major constraint limiting BIST from being a perfect single solution to all major testing concerns at the device level is that since the test implementations are internalized to the target design, the pin buffers and drivers of the device cannot be directly exercised. Internal loop back structures can enable transmit and receive functions to be verified through a substantial portion of the device but some sort of external provision must exist to exercise that final portion of the design. External loopback provisions in a test fixture could be applicable if the BIST circuitry is capable of evaluating the signals or some sort of functional test system may be required.

2.2 Alternate Test Improvements and Methodology

BIST and scan solutions are very powerful for enabling economically affordable test capabilities, but there are many applications, especially high-performance ones, that require functional testing. Unfortunately, while ATE systems are very powerful and flexible they can also be performance limited relative to newer devices very shortly after purchase due to the rapid pace of new technology development. With a very hefty upfront purchase cost, including the test mainframe, cooling infrastructure, control computer, and per-pin cost, it can be very expensive to consistently re-purchase the newest systems to keep ahead of the technology curve assuming the systems are even capable of the needed test functionality. Per-pin architectures are very flexible on an

individual channel programming basis and make the system very easy to expand since the constituent components are by definition individual which makes for easy linear expansion. But since each pin is independent of the others, some features are not possible, such as true differential signaling and dynamic signaling protocols which require analysis of signals relative to each other across pins.

Some additional design, methodology, and hardware innovations that have been designed to improve coverage, reduce costs, or implement previously unavailable functionality are presented below.

2.2.1 ATE Extension

There are a number of applications, especially with regards to those requiring very high bandwidth and source-synchronous signaling, in which neither the native capabilities of an ATE system or a similar test solution are capable of providing the needed performance. To meet these higher performance or formatting requirements, extension modules can be added to an ATE to extend the capabilities of the underlying system [10]. This is achieved through the use of driver and receiver modules integrated on the application load board between the ATE and the DUT (Figure 2.5). These modules can buffer, switch, or multiplex a collection of stimuli signals from the ATE to an alternate configuration or higher data rate and subsequently applied to the device under test (DUT). The corresponding receiver modules can reverse the process, adapting DUT originating signals back to a compatible format compatible to the ATE or demultiplex a fast data stream to a rate capturable by the ATE.

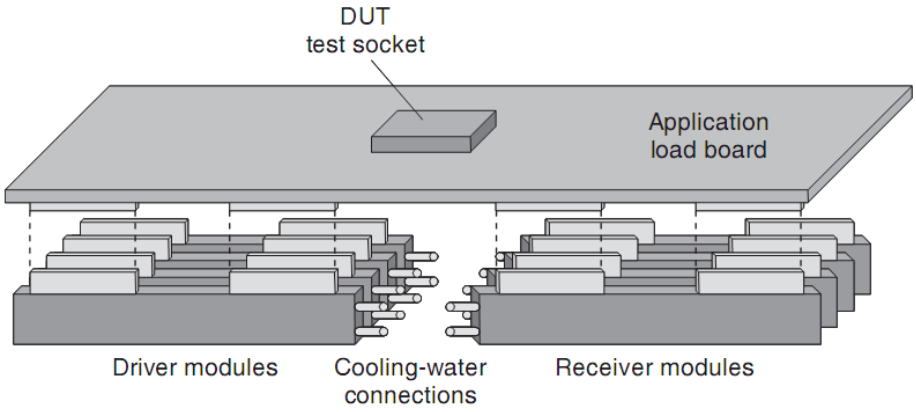


Figure 2.5 Intermediate driver and receiver modules extending the capability of the ATE.

For example, a test engineer can trade off moderate speed ATE channels in a high channel count tester to achieve a smaller quantity of signals at a signal rate higher than the base capabilities of the tester. Additional features not directly supported by the tester can also be implemented, such as the addition of switched relays to allow for loopback configurations, enabling a DUT to drive itself. Another application for these extension modules is to adapt a signaling protocol that would otherwise be incompatible with the ATE due to the bit-rate, the non-deterministic nature of the signals, or the requirement for DUT driven clock recovery [11]. Figure 2.6 shows how these extension modules affect and fit into the modified test signal flow. While the figure shows serializers/multiplexers which would be appropriate for a driver module, de-multiplexers could be used on receivers, relays for loop-back configurations, and many more options as individual configurations or complex combinations depending on the desired capability and acceptable complexity.

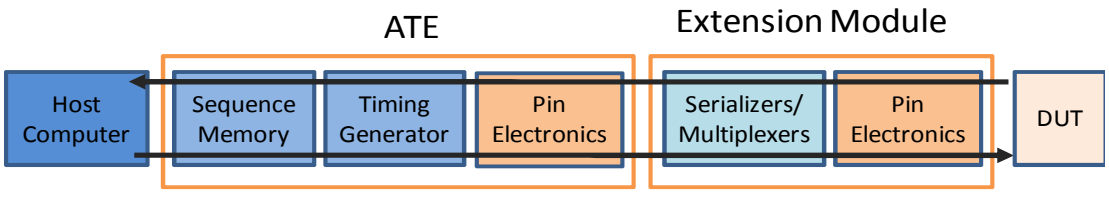


Figure 2.6 ATE stimulus and response data flow per signal pin.

2.2.2 FPGAs for Test Applications

Field-programmable gate-arrays (FPGAs) are devices consisting of large arrays of reprogrammable logic blocks. Coupled with dedicated hardware blocks for more complex logic elements such as clock distribution networks, block memories, processors, I/O blocks and even high-speed transceivers, a wide diversity of application logic can be generated and integrated into a single device. Designs can be created at the gate level or described in a hardware description language (HDL) such as VHDL or Verilog. Figure 2.7 shows a single slice, of which there are two per configurable logic block (CLB), from a Xilinx Virtex5 FPGA [35]. Each slice in the device contains four look up tables (LUTs), visible on the left, which can be configured to perform Boolean logic operations based on six distinct inputs and two outputs. The outputs of these LUTs can be routed through a variety of function multiplexors and optionally stored in memory elements configured as flip-flops, latches, or registers.

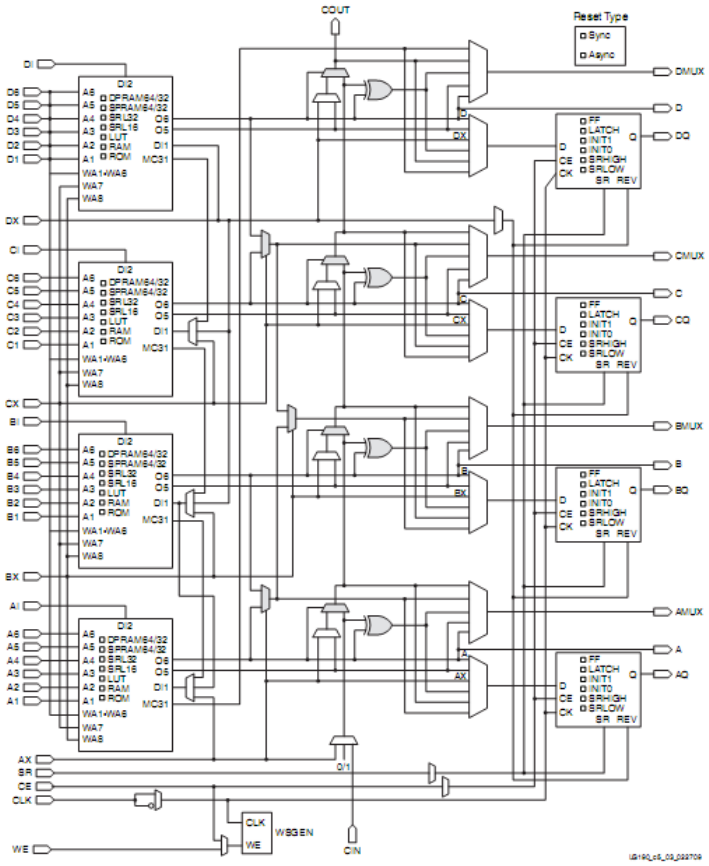


Figure 2.7 Slice diagram from Xilinx Virtex5.

The use of FPGAs for testing applications is increasingly favorable due to the channel density and highly integrated set of features supported by modern FPGA designs. FPGAs are especially useful for implementing large blocks of combinational logic within a single device, a task which would have previously required dozens of discrete chips. A digital designer employing an FPGA in a hardware design has available, even in the smallest and simplest of modern FPGAs, dozens or hundreds of I/O pins, hundreds of thousands of equivalent gate operations, memory blocks, DSP blocks, and more depending on the specific product model and manufacturer. These resources are available in a nearly unlimited set of configurations, programmable through the use of schematic capture, hardware definition languages, or the implementation of pre-configured intellectual property (IP) cores [36].

The use of FPGAs for testing applications or being incorporated into a tester infrastructure is not an innovative concept exclusive to this work. One solution, proposed by the FPGA manufacturing company Altera, is to utilize FPGA resources to implement sequencer, pattern generator, and data analysis functions as they would normally appear in an ATE infrastructure. These roles and explicit implementations could be varied through a modular IP core library (Figure 2.8) with the resultant design applying signals to and capturing responses from the device under test directly or through a set of external analog pin electronics. This approach leverages hard implemented and synthesized logic to achieve the necessary processing [37] to evaluate the system response. However, such a design is essentially an FPGA based ATE system and is therefore subject to all of the same performance and interface limitations previously presented, especially with regards to the limited capacity of the pin electronics to capture non-deterministic source-synchronous signals in burst transmissions.

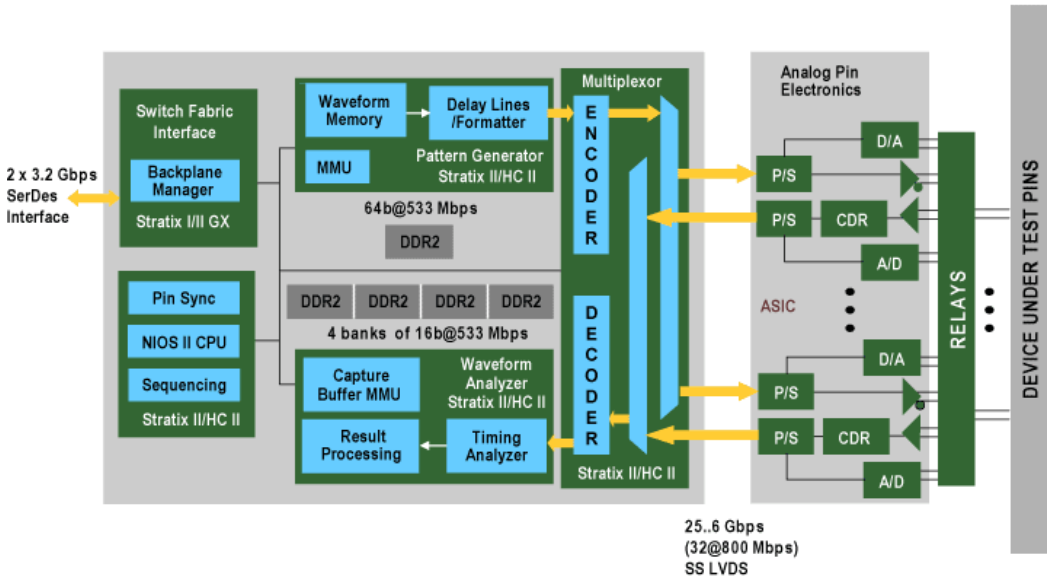


Figure 2.8 Altera Stratix device for ATE applications.

2.2.3 Digital Logic Core

With the versatility and ability to dynamically reprogram FPGAs, it is possible to create a general-purpose and reconfigurable logic circuit designed to implement many of the functions of traditional ATE. One such design is presented in [38] and forms much of the foundation for the work presented here. Utilizing the I/O pins of an FPGA in place of or in conjunction with the pin electronics of an ATE, a Digital Logic Core (DLC) (also referred to in some publications as a Digital Test Core) can be utilized to enhance the performance of the host ATE, provide additional test functionality, or operate as a standalone test instrument.

A diagram of a basic DLC structure based on a Xilinx Virtex5 FPGA is illustrated in Figure 2.9. A number of DLC variants have been constructed as part of this work and parallel works within Prof. Keezer's research group, incorporating many other FPGA models from Xilinx including VirtexE [39], VirtexII [1], Virtex4 [12], and most recently Spartan6. Some of the details and specific configurations may vary depending on the core FPGA utilized and the desired performance goals, but the general configuration remains consistent regardless of the exact model or even vendor of FPGA. The Virtex5 FPGA utilized in the DLC for the presented work is a 665 pin device, of which 400 are available user I/Os each capable of running up to 800 Mbps. The FPGA supports 2400 configurable logic blocks, 1152 Kbits of onboard memory in 32 dual-port capable memory blocks, and a hard-IP implemented PCIexpress processing block that can be used in conjunction with one or more RocketIO high-speed transceivers to implement PCIexpress connectivity with a minimum usage of general resources [40].

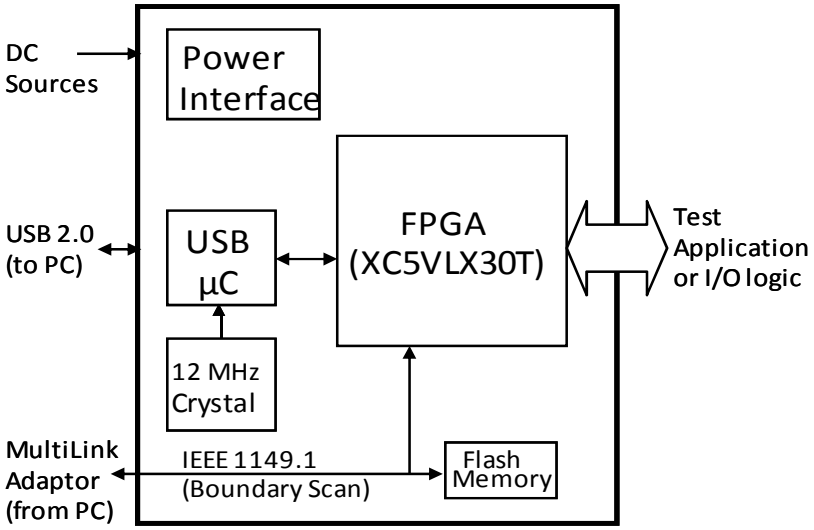


Figure 2.9 Digital Logic Core used for several test applications.

In addition, the DLC includes a specialized microcontroller chip for interfacing to a Universal Serial Bus (USB) host to support general data or configuration transfers to and from the DLC to a host computer. Supporting these are a 12 MHz crystal oscillator for USB communications, and a flash memory to store the FPGA programming information. The flash is programmed from a personal computer through an IEEE1149.1 (boundary scan) interface. Once programmed, it loads the personalization data to the FPGA upon power-up. The USB (or similar host computer data interface) and boundary scan programming links are optional and not required for all operational configurations. For instance, if a pre-configured test routine is programmed within the flash memory, download will occur from the flash memory to the FPGA and test execution can begin immediately upon power-up without the need for any additional external data connections, allowing for an autonomous and standalone test. Alternatively, the FPGA can be updated with a new design by direct programming or indirectly through a rewrite of the flash memory, enabling a single test system to be reused and quickly adapted for new test applications, or to make modifications in an existing design.

The pin I/O resources of the FPGA can be utilized directly in conjunction with the DUT or can be processed through intermediate circuitry, such as the extension modules as presented previously (Figure 2.10). These modules can be implemented as explicit circuits on the same interface board as the DLC or as reusable and removable plug-in boards. Test vectors can be stored within the FPGA memory resources, in secondary memory elements, or synthesized in real-time by state machines encoded in the FPGA.

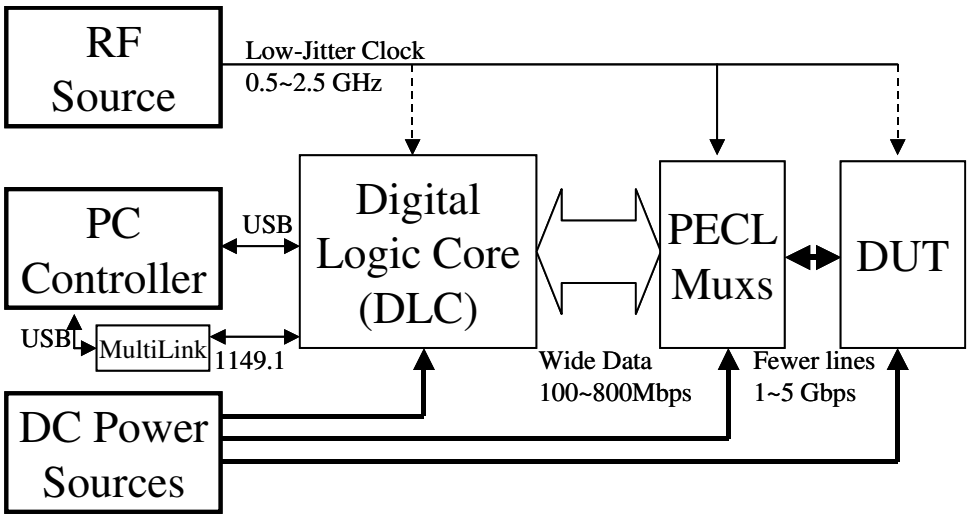


Figure 2.10 Using a programmable Digital Logic Core with high-speed PECL for testing a multi-gigahertz DUT.

The core capabilities of a tester constructed using a DLC and extension modules are very similar in principle to that of a much larger and expensive ATE system. The feature set provided by such a tester may be limited with respect to the wide range of features provided by ATE, but it can be tailored to provide just the specific test features needed for a particular application [1]. One application explored for these FPGA based Digital Logic Cores is in the form of a Test Support Processor (TSP) intended to enhance the native capabilities of an ATE [41]. Situated in the signal flow between the ATE and the DUT, the TSP can be utilized as an intermediate buffer situated physically much closer to the DUT than the respective ATE pin electronics. The generation of high-speed signals supported by the TSP would therefore have a much shorter distance to travel to

the DUT minimizing the design challenges of maintaining a well-shielded, impedance-controlled signal path over the long distance the signal may have to otherwise travel.

The natural evolution of this design is to implement many of the features provided by the ATE directly within the test core, allowing the design to function independent of the larger ATE infrastructure. Through this approach, a miniature tester or even an array of miniature testers can be employed where the economics or physical size of a fully fledged ATE would be impractical or infeasible. A prototype system for the purpose of wafer-probe testing of wafer level packaged (WLP) is presented in [42] and is shown in Figure 2.11. The resulting system (left image) is as previously described, a DLC incorporating a Xilinx VirtexE FPGA, augmented with external positive emitter-coupled logic (PECL) circuits for timing adjustment and data multiplexing to higher signal rates. The target application is the parallel testing, assuming an array of these DLC based miniature testers, of a wafer of devices before they have been cut apart into individual units.

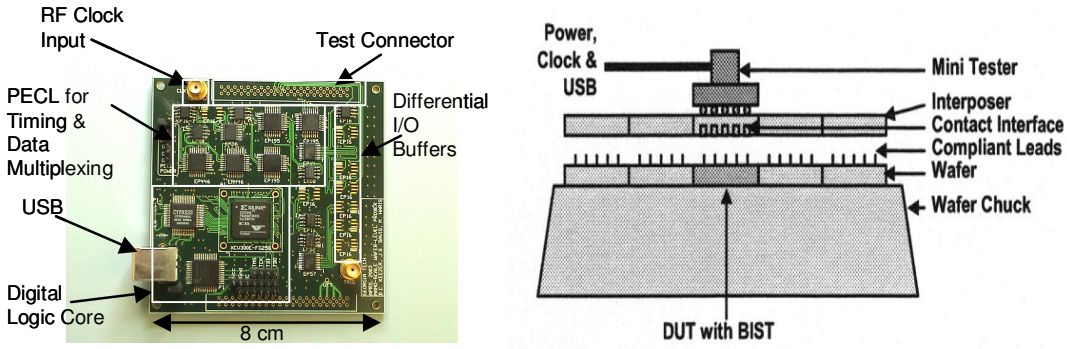


Figure 2.11 High-speed wafer-probe testing of wafer-level packaged devices using a DLC based "miniature tester" and a high-density interposer.

2.2.4 Protocol Aware ATE

As the above designs show, the use and incorporation of FPGAs into some sort of test infrastructure is not a particularly new or novel approach. However, the idea of integrating an FPGA or similar reprogrammable logic within the test path with the intent

to dynamically react to the ongoing state of the test or implement tester interaction with the device under test is a relatively new one. Developed in parallel to this presented work, Protocol Aware (PA) ATE explores the idea of test systems designed with the capability to natively emulate real time chip interfaces at the pin or pin group level. PA-ATE is a proposed evolution of ATE architecture to allow for true system level test [14]. Conventional ATE are difficult to interface with increasingly prevalent source-synchronous systems and are also incapable of native and real time handshaking with a variety of data transport and debug I/O protocols.

Some of these protocols, such as JTAG, PCI, SPI, I2C, or I2S, are widely utilized for system and diagnostic purposes, including DFT and BIST analysis discussed earlier. This protocol incompatibility exists because of the signal flow utilized for each pin of an ATE. Each pin operates independently of other pins, receiving a pattern sequence and timing information from a controlling host computer. When the test sequence has been downloaded to all of the system pins, the signals are transmitted through the pin electronics to the DUT or, as presented earlier, can be further modified through an extension module which may multiplex that single pin with many others to achieve a higher data rate signal [43]. In a conventional ATE, the signal sequence is pre-determined and is not dynamically altered over the course of the test execution. If the DUT implements a protocol which requires a handshake to occur as part of the process, all of the signals and the timing must be evaluated and incorporated as part of the signal sequence. This is most likely achieved from simulations run from a synthesis model of the system created during the design process or can be modified manually.

Figure 2.12 shows the flow of a traditional testing cycle. The process is iterative, starting with an HDL language model of the DUT. This model can be simulated and the resulting signals processed into generic logical representations of the stimulus and response vectors to be generated or captured by the tester. These vectors can be further compiled into test data files specific to the tester of choice. The tests can then be

executed and the results analyzed. In a production environment, these results would correspond to a pass or fail with respect to accepting or rejecting the system or device from a manufacturing standpoint. In a development environment, the resulting vectors from the test can be processed and evaluated by a test engineer to further refine aspects of the initial simulation. At this point the cycle can be repeated. Because the test vectors are giant arrays of ones and zeros, achieving a small logical change which should be simple, such as a reset, reinitialize, or a tweak to a data value, may not be achievable directly on the tester and instead must be made in the simulation and propagated through the process [44].

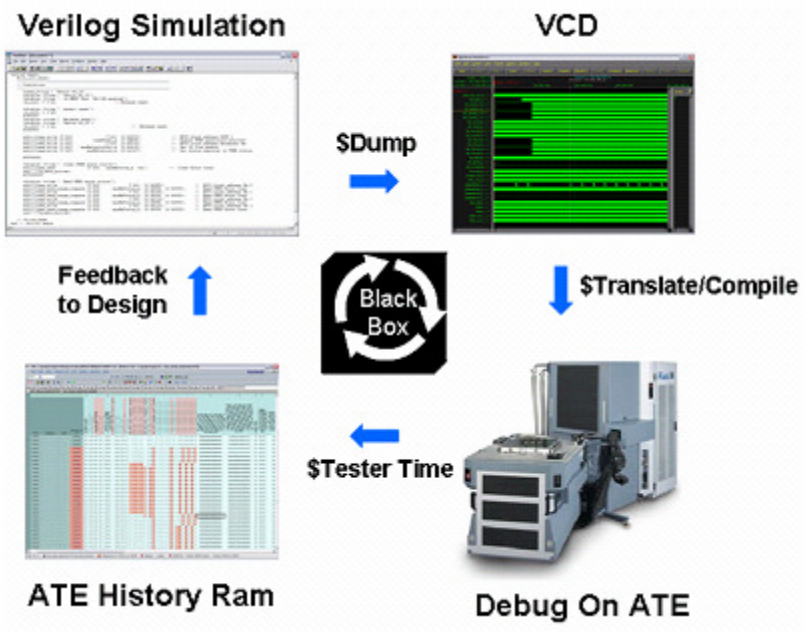


Figure 2.12 Traditional test iteration loop.

A PA-ATE implementation can be achieved through a variety of means. The simplest and easiest way to verify the validity of the approach is to create a virtual emulation, utilizing the controlling workstation of an ATE to process and modify the signal streams through a simulation and synthesis process. This is essentially the same process illustrated above, just automated and designed to reactively respond to the ATE output and adjust the input test vectors accordingly. This unfortunately results in very

large latencies, as data must be offloaded from the tester system, processed, and reintroduced to the tester.

A hybrid approach, utilizing much of the existing ATE infrastructure and including an additional protocol aware processing element, is a much more desirable approach that can result in greatly improved latencies and therefore increased compatibility. This hybrid design approach would incorporate an FPGA or processing elements of an FPGA (which could be shared across pins or pin groups) immediately following the pin electronics which interface to the DUT (Figure 2.13). Tests could optionally be executed in a traditional fashion, simulated through the host computer, or the logic could provide a lower latency bypass route capable of returning a protocol request to the DUT based off state transitions or stored memory sequences. The use of this design would greatly reduce the latencies required to process the protocol dependent transactions [45] but would not eliminate them completely. The merits, implementation challenges, and feasibility of this design approach are still being debated [46-50]. Some commercial solutions, such as the Protocol Aware Test System KT-7200 FINN, have recently begun to emerge [15].

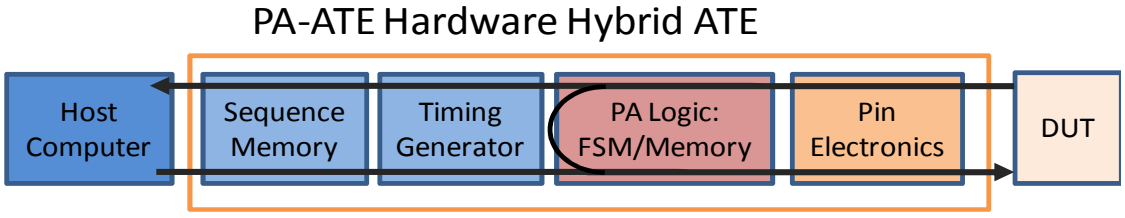


Figure 2.13 Protocol Aware modification to standard ATE signal flow.

This rearrangement of processing roles changes the fundamental test flow when using PA-ATE systems. Where the test process was an iterative loop before, development stages in the modified test process, shown in Figure 2.14 [14], become much more dynamic and bi-directional. There is a more direct correlation between the HDL simulation and the design interactions, allowing for ongoing tests and analysis from either the bench or ATE to be utilized to improve the tester integrated HDL

implementation. Also, since the HDL implementation handles the majority of the signal synthesis, bench analysis and ATE feedback can be utilized equally to enhance the model as the intermediate steps of compiling tester specific signal vectors is eliminated.

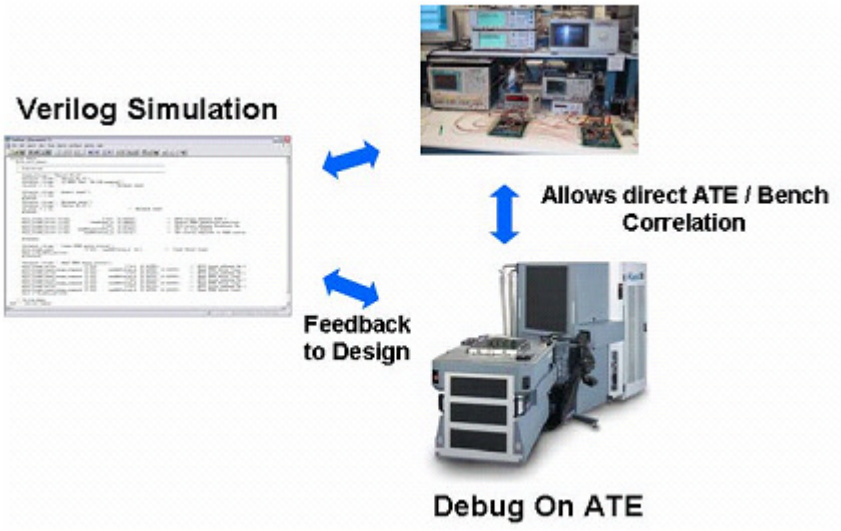


Figure 2.14 Dynamic test flow.

2.3 Non-Deterministic Signaling

An ongoing change currently occurring at the system-level is the transition from parallel synchronous signaling to source-synchronous signaling. These low voltage, differential, clock embedded, and point to point signaling methods have been widely utilized in telecommunications and the testing of these systems is fairly simple due to the use of relatively few channels at a time and the implementation of commercially accepted standards and protocols. Due to the low channel count and standardized signaling, instrumentation-based testing is an acceptable test methodology for these systems [51].

Newer computer systems adopting source-synchronous signaling techniques however do not necessarily fit this same model, employing wider channel counts and may employ proprietary signaling protocols. Testing of these systems using ATE is complicated due to the very nature of the signals which utilize very high data rates

(multi-Gbps), DUT driven timing, and signals that are nondeterministic in time [11]. The traditional test flow is predicated on the basis that there is a specific and consistent relationship between the tester and the device under test, but non-deterministic signaling interfaces violate this relationship. Conventional testers natively designed or augmented with internal or external add-ons to support many channels, each supporting differential, source-synchronous channels with variable latencies, do not exist or would be too costly if they did [11,51]. One possible solution is the use of embedded built-in self-test (BIST) capability and design for test (DFT) techniques to enable the device to evaluate its own state or to eliminate the non-deterministic aspects of the test stimuli and response vectors [52]. Another solution is the augmentation of the ATE system with additional test modules specifically tailored to the I/O protocols required [11].

Another difficulty with testing and data recovery in a system utilizing source-synchronous signaling is the recovery of data in bursty or packetized segments. These types of transmissions in a synchronously clocked system are no different than a continuously transmitted equivalent. However, due to the embedded clock nature of source-synchronous transmissions, a destination receiver is forced to compensate for the phase of the incoming signals to utilize the embedded clocking features. One method of embedding this clock is through the use of the edge transitions within the data stream itself. A common interface bus that uses this form of signaling is PCI Express [53]. A clock and data recovery (CDR) circuit can align to these edges and synthesize or phase lock a local clock to sample and recover the data stream. This process is lengthy however and a lock cannot be guaranteed for hundreds or thousands of cycles [54,55] after the initiation of a link, requiring any such transmissions to be prefaced with a segment of data intended to 'train' the destination and achieve a locked clock. This preface limits the minimum packet size in a switched environment and can prove to be a large source of overhead if the overall packet is proportionally small.

An alternative to the use of CDR circuitry is the use of an additional channel, beyond those used for data, as a dedicated clock. A common interface bus that uses this form of signaling is HyperTransport [56]. As long as this parallel transmitted clock signal follows the same path as the data signals, the clock-to-data skew should remain consistent and allows for the near instantaneous recovery of the data at the destination. Some devices require a small setup period after inactivity or a forced reset [57] but this is much faster than a comparable CDR implementation.

2.4 Desired Tester Functionality

From the systems and design solutions presented above, a list of desired features and function parameters can be created. To test a specific system like the Data Vortex and to ensure viability for future designs, a test system constructed using this architecture needs to be capable of:

1. Multichannel, high-speed (multi-Gbps) serial signaling
2. Source-synchronous signal capture
3. Non-deterministic signal capture and processing
4. Emulation of protocol specific interfaces

It was determined that the best solution to provide all of these requirements was the integration of an FPGA, leveraging the reprogrammable logic and flexible general purpose I/Os, and high-performance application modules. The application modules serve a similar design role as the ATE extension modules discussed earlier, extending the capabilities of the underlying system to support signaling rates, signaling formats, or other functionality that would otherwise be unavailable without those modules. Multiple high-speed, source-synchronous serial channels for both transmission and reception can be implemented using serializers and deserializers in conjunction with the numerous general I/O available on most FPGAs. The FPGA logic, in conjunction with application

specific module solutions, can be used to resolve non-determinism in received signals as well as respond to protocol specific states.

The design approach utilized in this research achieves many of the design improvements as proposed through PA-ATE but through a different architectural approach. Where a PA-ATE implementation as described in [14] would be reactionary and processed on a pin-by-pin or pin cluster basis, this solution applies the concept at the full system level. The modified test flow in this research, as compared to that of the traditional ATE flow presented in Figure 2.1, is shown in Figure 2.15. The largest change is the dissolution of individual sequence memory and timing generator elements which are replaced by a larger general purpose block of logic within the FPGA capable of sequencing data, generating patterns, and analyzing results.

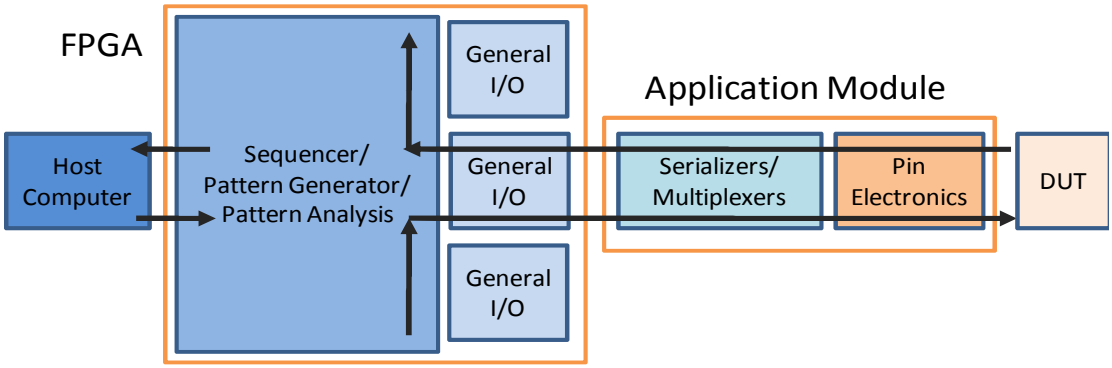


Figure 2.15 Test flow with dynamic pattern capability.

2.4.1 Test Signal Generation

With respect to signal generation, the tester in this research needs to conform to most of the design principles described above for functional test systems. Signal sequences must be accurate in both value and timing to match to the target DUT. Most FPGA general purpose I/O pins support a range of voltage specifications and time shift capabilities in very coarse increments. Default signal encoding is NRZ. If the desired

signal performance is within these base ranges, the FPGA signals can be used directly. Otherwise they can be enhanced through some sort of external logic, adjusting the electrical range, a shift in timing, or encoding format. These features are generalized in the test flow figures presented above as 'pin electronics.'

A subset of signal enhancement touched on previously is serialization or multiplexing. This is the process of taking two or more signals and interleaving them in time. The primary purpose of this approach is to create a combined signal that is at an effectively higher signaling rate than the source, or the FPGA in this case, would be capable of creating. The most basic example of this process is often referred to as double data rate or DDR signaling. For DDR signals, edge transitions occur relative to both rising and falling edges of a source clock thereby doubling the overall rate. Serializer devices extend this principle, taking N slower rate signals in parallel, typically 4, 8, or 16 but can vary by model, and convert that collection of signals into a serial equivalent at N times the original signaling rate. For example, the transmitter modules used throughout this work take eight 312.5 Mbps signals generated by the FPGA and convert to a 2.5 Gbps serial stream utilizing a 1.25 GHz reference clock. In this case the serializer itself utilizes DDR to operate off a reference clock running at one half of the final signal rate which reduces the design burden of distributing a higher frequency clock throughout the design. These elements are discussed in more detail in Chapters 4 and 5 with respect to the test development platform and the transmitter plugin modules respectively.

Signals can be generated by the FPGA as programmatically generated sequences or pulled from a pattern memory populated from a pre-generated sequence. Regardless of how the signals are generated, any application modules or enhancement circuitry that has been added to the system beyond the FPGA must be accounted for. A secondary relationship is established between the signals as generated at the FPGA I/O boundary and what is actually applied at the DUT. These changes can be pre-evaluated and incorporated as part of the original test vector creation, just as would be required if those

same modules were used in conjunction with an ATE. The combinational resources of the FPGA can be utilized for processing the data, either as an intermediate translator to accommodate the bit-stream for the modules or synthesize from the ground up.

2.4.2 Test Result Capture

The capture of test results is the logical and complimentary process to generation. Signal capture involves sampling response data from the DUT on the receiver side of the test system. Data rates beyond the native capability of the FPGA must be converted down and is typically referred to as deserialization or inverse multiplexing. The capture process can operate at full rate, converting serial streams into N parallel bit equivalents at $1/N^{th}$ the original signaling rate. Alternatively an undersampling process can be utilized which can require much longer to create a comprehensive representation of the sampled signal but may be more inexpensive to implement or necessitated by design constraints. Modules to enhance the fpga signal limitations. Undersampling solutions may also be insensitive to transient or intermittent errors which may occur outside of the active sampling region.

Complicating the capture challenge is that this system must be capable of receiving source-synchronous signals. Source-synchronous signals have a clock relationship to a parallel transmitted clock or to clocking information embedded within the data stream itself. Many test systems, ATE especially, rely on a tester sourced clock or an external clock synchronous to all connected system elements. Neither of these situations are directly compatible with source-synchronous signaling. One solution to this problem as presented earlier is the use of extension or application modules specifically designed to adapt source-synchronous or asynchronous serial streams into a synchronous format more compatible with the underlying test solution. The proper choice of a deserializer device can allow for the creation of a synchronous relationship

with the destination device, in this case the tester FPGA. These features and receiver module design are discussed in more detail in Chapter 6.

Once the incoming signals are captured, processing and analysis of the system response can begin.

2.4.3 Processing and Analysis

In addition to signal generation and capture, some form of processing and analysis capability is required to complete the process of validating the DUT. In the simplest form, analysis can be performed virtually identical to standard functional test systems, performing a literal comparison of expected results to the captured response. Any of the methods mentioned, including synthesis from a model and stored in memory, a signature analyzer [58], or emulation logic implemented in the FPGA based on the above model, can be utilized. The resulting system could in principle and with the right programming and performance modules replicate the behavior of a per-pin style ATE.

However, to fully satisfy the design requirements for non-deterministic and protocol specific interfaces, a more in depth processing capability needs to be available. Through the use of an FPGA and the programmable logic resources, a system can be constructed capable of dynamic and reactive response, providing feedback to the signal generation systems or adjusting the capture process to implement protocols or features that are impossible at the bit level alone.

For instance, the principles presented in the discussion of ATE extension, specifically the use of serialization and deserialization circuitry to convert large quantities of parallel data into a higher-speed yet more compact collection of serial streams, is a technique critical to the test modules in this research. Consider a received serial stream that is deserialized and converted into a byte wide parallel data word for processing at a data rate much lower than that of the serial stream. The exact methodology used can vary, but the principle is simple: collect eight sequential samples at the serial stream data rate

and present them on a secondary interface at one-eighth the data rate as an eight bit parallel word. One possible implementation is the utilization of a shift register with parallel read capability, sequentially shifting in samples from the serial stream and parallel loading the data into a secondary register every eight cycles. Another implementation could use a set of eight undersampling circuits, each successively out of phase from the previous by one serial bit period.

Both of these implementations are susceptible to the same general failure which is related to incorrectly beginning the signal capture process with the first bit of the desired eight bit sequence. Of the two implementations described, the problem may reside in improper synchronization with the eight element counter or a timing mismatch to the first undersampler, respectively. However, the net result will be the same. Instead of capturing the expected bits in the sequence expected, one or more bits from a previous or subsequent byte sequence will be present and all other bits will be shifted by a similar number of places. This flawed sequencing is shown in Figure 2.16.

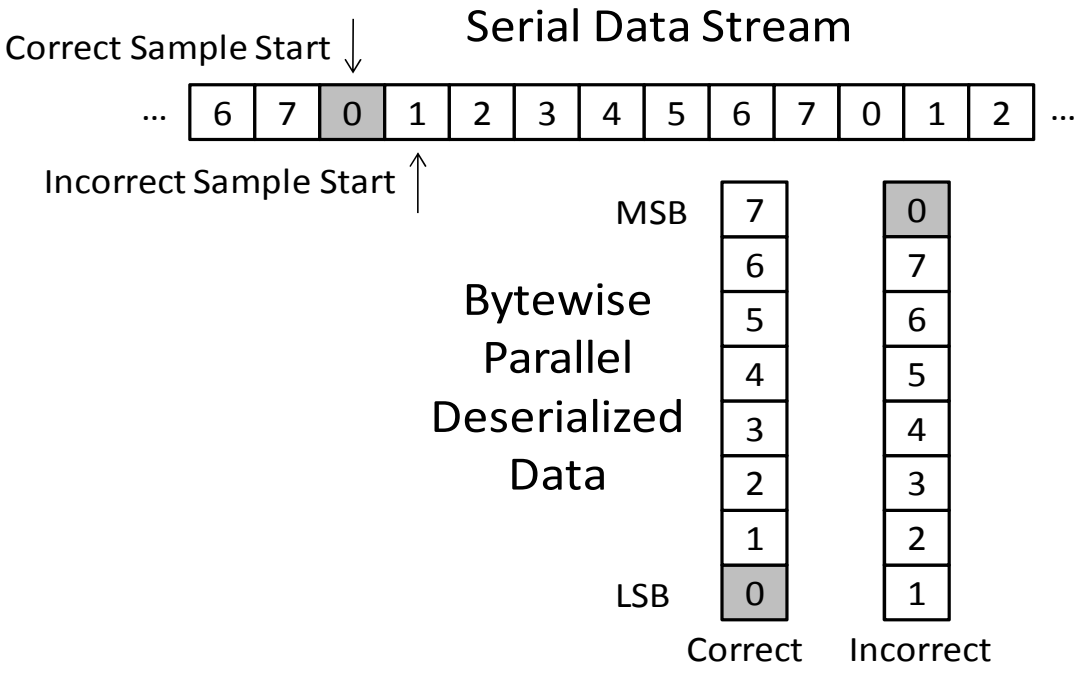


Figure 2.16 Deserialization sampling skew example.

In a non-dynamic tester, should such a timing misalignment occur, all of the data collected in this fashion would be deemed incorrect and the test would fail. Depending on the device or system specification, this failure may be literal or it may be a false positive. If this 'failure' mode was understood ahead of time and recognized by the test operator, the timing could be adjusted and compensated for on a part by part basis to eventually achieve a pass. Or if this timing relationship was consistent across all the devices or systems it could be globally accounted for as part of the characterization process. However, for source-synchronous and time-variant systems which are the test focus for this research, these timing inconsistencies are fundamental to the designs and may not be consistent.

To account for this, there must be accommodations in the tester implementation to resolve this dilemma and others like it. There is no one perfect solution to these problems, as this specific issue has at least two solutions. The first one has already been discussed: correct the timing alignment. The potential solutions to correct this timing relationship depends on the hardware implementation but the net result will require the serial data stream to be skewed in time relative to the clock driving the deserialization logic. This can be achieved by directly skewing the incoming data stream or the clock. Under some circumstances, if there is a direct causal relationship between the stimulus signal being generated by the test system, this skew can also be achieved by altering those signals. If a shift and count deserializer is employed, the timing relationship can also be forcibly altered by resetting the counter synchronously to some known time reference or causing one or more shift cycles to be ignored until the proper alignment is achieved. A second possible solution to this particular problem is to utilize the processing resources available to the system, either the FPGA itself or the control software, to recognize and manipulate the data to remedy the alignment.

CHAPTER 3

PRIMARY TEST BENCH

The Data Vortex interconnection network is an all optical packet switching (OPS) network built upon a novel switching topology that is being developed at Columbia University, intended for ultrahigh capacity, low-latency transfer of small data packets in massively parallel computing applications. The nature of this OPS network and the testing challenges associated with evaluating the exchange of messages across the system as well as the sourcing and capture of simulated traffic through the network taxes the instrumentation resources of an entire lab. Many experiments demonstrating injection and reception of packets utilizing a variety of routing paths, collision avoidances, signal quality, and payload capacity have been conducted [59,60]. Unfortunately, comprehensive evaluations of the network utilizing simulated (or actual) point-to-point message exchanges have not been conducted because the physical systems required to generate and process the respective signals to do so do not exist. The necessity to create a custom test system capable of natively interfacing to the network, employing very short duration packets utilizing high-bandwidth, dense wavelength multiplexed (DWM) payloads that arrive asynchronously depending on routing path is one of the guiding motivations for this research.

Though the test platform developed in this research has been designed to support a wide range of applications, many of the design parameters of the test solution described in this work are derived from the testing requirements of the Data Vortex network. The network contains features which are difficult to interface to and test utilizing conventional techniques and equipment. The two greatest test challenges are inherently related to the packetized nature of the network, namely the use of very short duration packets at 25.6 ns in total length and which arrive in bursts. The assembly of signal

packets requires a specific set of combinations of signals conforming to very tight timing parameters. Capturing the data at the destination is also difficult since the connection is routed on a packet by packet basis. This prohibits the establishment of a virtual connection between the source and destination that could be used to synchronize the two ends of the connection. Distributed clocking is also unfeasible because the packet arrival is asynchronous due to the routing mechanisms used within the system combined with minor length mismatches between individual network segments. The final design challenge is the very short duration and number of bit transitions within the packet which prohibit the use of clock and data recovery from the signal stream itself. This prohibits the use of conventional CDR techniques which would be used in a comparable but longer duration signaling system as there is not enough time to lock onto the phase and recover any data before the end of the packet.

3.1 Architecture

The architecture of a 12x12 port version of the network is shown in Figure 3.1 [60]. The topology is described using three parameters: cylinder, angle, and height. The input signals are injected at the outermost cylinder and travel one angle forward within each packet time slot, either forward along the same cylinder or towards the next innermost cylinder. Routing occurs within the same cylinder if the appropriate height has yet to be reached within that cylinder, or if the node that the packet would occupy if deflected along the inner angle is already occupied [61]. This switching behavior prevents node contention from ever occurring and creates a virtual buffer, circulating signals that cannot be routed inwards at the expense of blocking new packets being accepted at the input ports. The packet will continue to circulate through the system until the appropriate destination node on the innermost cylinder is reached or until the signal no longer has sufficient power to be recognized by a node (which should take 60 hops or more given the current node implementation) [62].

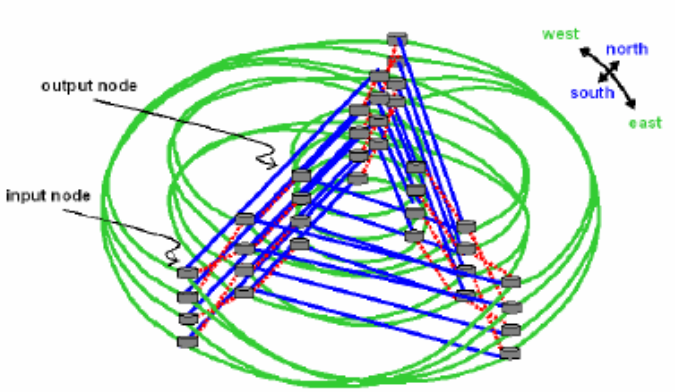


Figure 3.1 Source-synchronous optical packet switching network (12x12 Data Vortex).

The Data Vortex topology, upon which the target network is based, is an all optical packet switching network designed for low latency communications ideal for supporting large scale distributed shared memory computing. A control and routing signals embedded within the packet are used within the OPS network for routing decisions while payload wavelengths are transparently passed through the system without being processed. This architecture eliminates the need for complex optical buffering, and the parallel presence of the routing information eliminates the need to further process the packet payload to decode and evaluate routing decisions. A traditional electrical networking system would require buffering and retransmission of the signal at each routing node, as would older generation optical networking systems (which require additional conversion of the signal from optical to electrical form and back again). By avoiding these unnecessary buffering and conversion steps, the optical packet switching network achieves considerably reduced latencies compared to store and forward networks. In addition, since there is no conversion, sampling, and buffering of the payload, the performance of the network is independent of the number and signaling rate of the payload channels. This permits future expansion in both parameters without any necessary changes to the network itself.

The optical signals travel in extremely high bandwidth optical fibers, utilizing dense wavelength division multiplexing (DWDM) to transmit multiple parallel data streams together. Routing bits and payload channels are contained on separate and independent optical wavelengths, allowing the network to scale in payload capacity by varying both the number of payload channels as well as the signaling rate of those channels. At each switching node, optical components observe the routing bit wavelengths which are quickly converted to electrical control signals that switch the payload between paths. The payload data, however, is never sampled, converted, or altered within the network. In that sense the network is optically transparent to the payload data, allowing for an almost unlimited variety of payload formats and widths. As the packet flows through the Data Vortex it progresses closer to its intended destination. It is switched between concentric "cylinders" and between "levels" within each cylinder. The packet circulates around the cylinders progressing inward and to the destination elevation corresponding to its intended exit port, where it is passed through dedicated optical fiber to the corresponding receiving electronics.

3.1.1 Physical Design

Routing nodes within the Data Vortex operate as a 2x2 switch with two possible entry points and two possible departure paths (Figure 3.2 [60]). Thirty percent of the optical power entering the node is diverted to filter out the framing signal and the header bit being utilized at that node for routing. These signals are processed electrically to select the proper output path that the packet is to take. The remaining seventy percent of the optical power entering the node is evenly split between the two possible output paths which are gated by semiconductor optical amplifiers (SOA). Only one of these SOAs will be activated by the control logic progressing the packet inward through the system or continue it circulating around on the current cylinder.

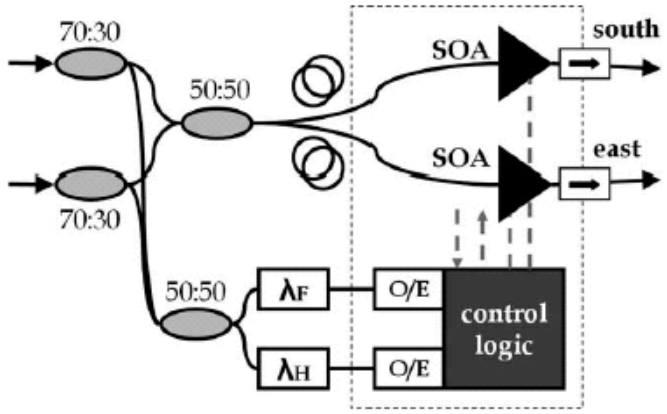


Figure 3.2 Switching node schematic for a Data Vortex node.

While packet injection is conducted synchronously across all of the source nodes, which are also referred to as injection nodes, arrival is asynchronous. The propagation time through the network is dependent on the specific path taken and, combined with minor variations of the segment lengths, means that the arrival of packets will have an indeterminate amount of time lag and phase skew relative to the source. As a result the payload must contain provisions for recovering these bursty source-synchronous signals at the destination, such as a separate payload channel that can be used as a clock. While certain fixed wavelengths are reserved within the system for routing and control information, only these portions are picked out from the main packet and converted to electrical signals for processing to make appropriate routing decisions (Figure 3.2).

Figure 3.3 [60] shows a photograph of a switching node from the Data Vortex. The passive optical components, such as the couplers and wavelength filters, can be seen to the bottom right in a plastic box. The frame and header photodetectors are to the left with the PECL decision circuitry in middle. The output SOAs are shown to the far right. The particular p-i-n photodetectors used for this board are rated for a minimum average power sensitivity of -26 dBm at 155 Mb/s. This low frequency operation range seems odd, but is more than sufficient considering the control signals actually operate at the injection rate of roughly 39 MHz. The SOAs are commercially available devices with a

noise figure of 6.5 dB, compensating for the coupling and connector losses of 5.1 dB along the packet path, and an unsaturated input power of approximately 0 dBm when operating at 50 mA. Only one of the two SOAs are driven at a time, due to the routing algorithm in use, meaning that the second unused path is blocked and prevents the packet from leaking into the second output port. The SOAs operate with a rise and fall time of approximately 1 ns.

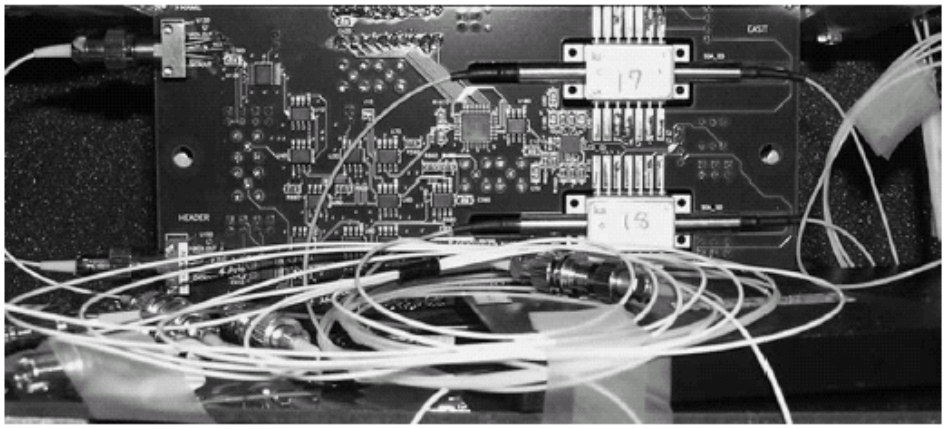


Figure 3.3 Photograph of a switching node.

All of the remaining payload channels are unprocessed by the system and routed jointly through the system, proceeding from hop to hop without being electrically captured and retransmitted. Because the signal does not undergo any intermediate electrical conversions, the content cannot be evaluated or altered internal to the system and prohibits the introduction of any internalized test infrastructure. The interfaces to this system are therefore only at the extreme perimeters of the system as a multiplexed optical signal or, as viewed further out beyond the respective opto-electric and multiplexing/filtering apparatus, as the respective demultiplexed electrical signals. Additionally, since the network is transparent with respect to the nature and quantity of signals present as the payload, the specific signaling used is a function of the transmitter and receiver modules, rather than any of the nodes implemented within the network. Assuming the timing and alignment requirements for the network (Figure 3.4) are adhered to, the signaling format of the payload signals can be modified as needed.

3.1.2 Packet Structure

The total length of a packet is determined by the time in-flight that the signal requires to clear a complete segment, including the switch components as well as the length of optical fiber connecting from one node to the next. These interconnecting lengths can be adjusted shorter or longer to accommodate a different length packet, but reconfiguration is a very time consuming and resource intensive process so the network has been tuned to a universal packet length of 25.6 ns (Figure 3.4). Due to a number of contributions including part variation, fiber length mismatches, and finite switching times of the output SOA drivers, a buffer dead time of 3.2ns has been incorporated between packets. This also ensures that slightly mistimed packets do not overlap and collide which would be catastrophic to both signals because, due to the all-optical nature of the network, such a collision would not be detectable nor could such status information be transmitted to the source or destination if it was detectable.

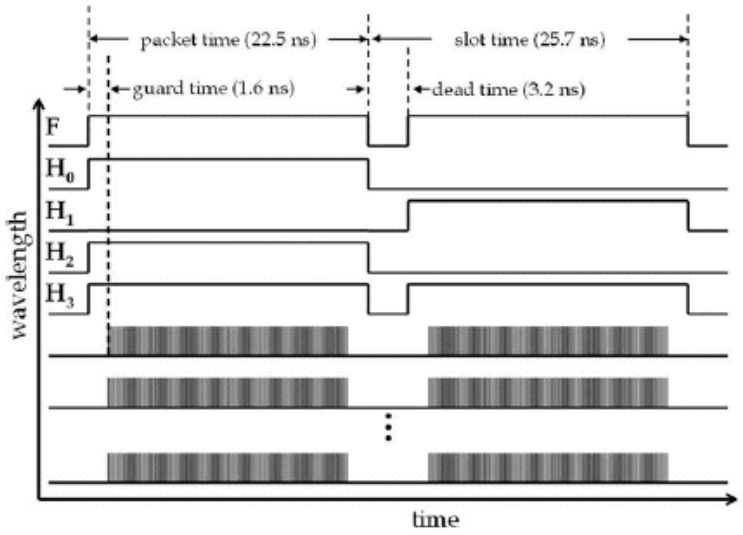


Figure 3.4 Wavelength-parallel packet structure for the Data Vortex with timing requirements.

Specific to the 2.5 Gbps hardware implementation, an extra data channel is included in the payload. This channel functions as a parallel clock for sampling of the data channels at the destination node. The clock signal is slightly longer, containing four leading transitions and six trailing transitions required by the deserializer circuitry for

setup and flushing of the device. The sizable guard times are required due to the routing nature of the data vortex as discussed above. The network does not buffer packets, instead deflecting them away from occupied segments. A virtual buffer is created by allowing a packet to circulate around a cylinder until the exit port or routing path is available.

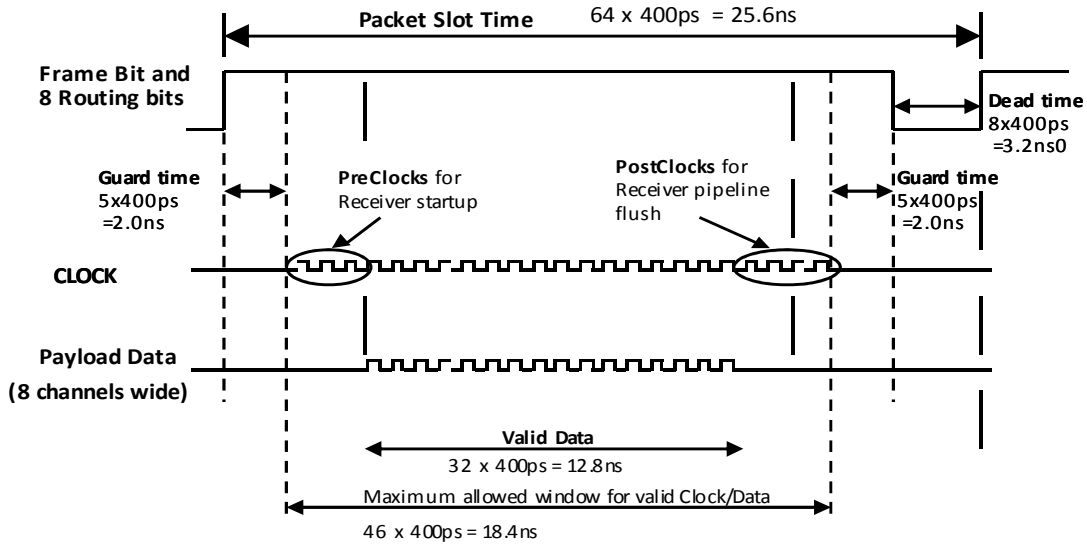


Figure 3.5 2.5 Gbps optical packet protocol for the Data Vortex.

Because of the wavelength specific filtering required at the switching nodes, the frame and header bits are encoded on the following fixed wavelengths:

Frame (1555.75 nm) - indicates the presence of a packet

H_0 (1535.04 nm) and H_1 (1533.47 nm) - two bits to indicate destination height

H_2 (1550.92 nm) and H_3 (1531.12 nm) - two bits to encode destination angle

Payload wavelengths are chosen from the C-band from 1536.6 nm to 1559.8 nm with 0.8 nm WDM spacing. These wavelengths are passed from source to destination transparently with no processing within the network. The optical spectrum of a packet within an early implementation of the Data Vortex is shown in Figure 3.6 [62] with some of the pertinent wavelengths labeled, including four payload signals BP[0-3], Frame F, and parallel clock CLK.

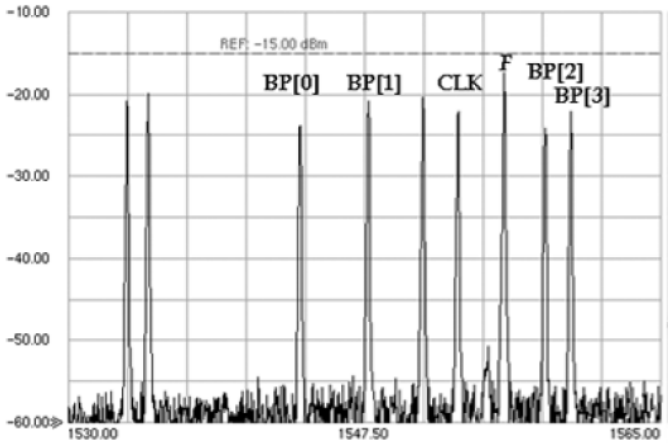


Figure 3.6 Optical spectrum of a sample packet.

3.1.3 Signal Flow

Figure 3.7 shows an example of the overall system configuration and signal flow utilizing a single tester. The tester generates electrical signals that must be converted an encoded onto distinct optical wavelengths. The individual optical signals are further multiplexed onto a single fiber interconnection per injection node and presented to the network. The network routes the signals to the appropriate destination based on the routing information contained in the packet where the procedure is reversed. Each wavelength corresponding to a signal required to be received by the tester, which is all but the original routing bits, is separated and converted back to an electrical signal and presented to the tester for processing.

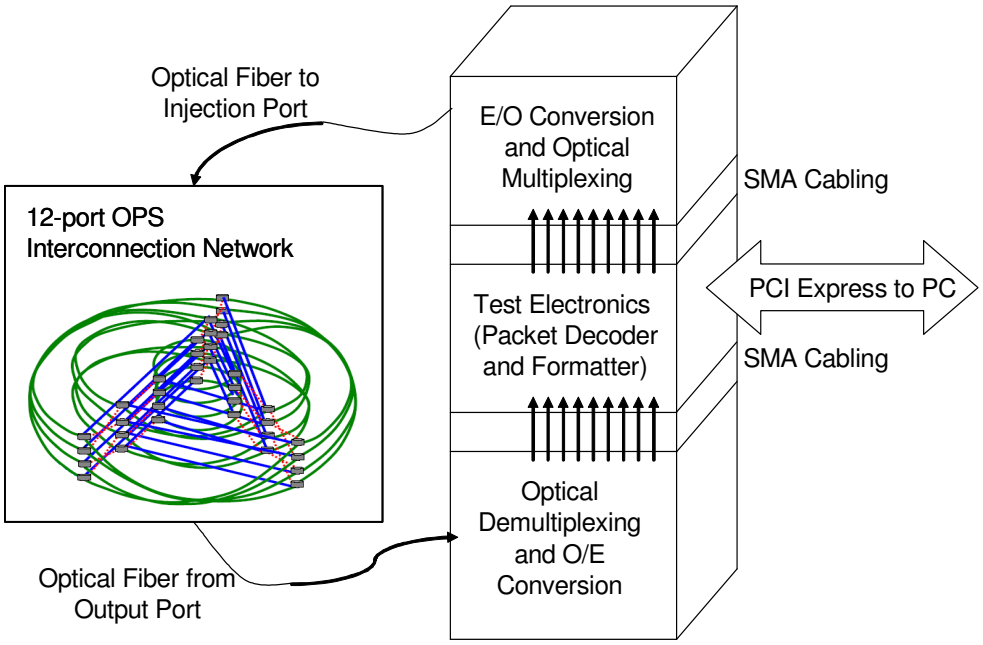


Figure 3.7 Overall system configuration and signal flow.

Since the respective systems are constantly under further development, the majority of the opto-electronic interfaces are modular, utilizing differential PECL signals over flexible cables and SMA connectors. The tester can be configured to operate as a standalone system or can be interfaced to an upstream data and configuration source. The existing design supports uplink interfaces over USB and/or PCIexpress though the design can be adapted as needed to any similar links as required depending on the latency and throughput requirements. The test electronics contain the components necessary to structure and transmit outgoing network packets as well as receive and process the incoming packets. Careful attention has been made to the selection of the electrical to optical (O/E) and optical to electrical (E/O) components to support the burst nature of the signals.

While Figure 3.7 shows the system in a loopback configuration, a single transaction across the OPS network is one way as shown in Figure 3.8. For example, a transaction originating at a computer is received by the test electronics, formatted into an aligned packet, augmented with the network control signals and time aligned, converted

to optical wavelengths and injected into a network input port. The packet circulates to the appropriate destination port where the Frame, Payload, and Clock signals are converted back to electrical form, deskewed (to partially compensate for any chromatic dispersion that may have occurred within the network or part variation between channels), processed by the test electronics and transmitted to the destination personal computer (or similar system, such as a high-speed memory node).

If a response is required, such as a memory read request, a similar but independent transaction would occur with the destination and source nodes reversed. For testing purposes a single board can mimic the respective transmitter and receiver components of the injection and egress nodes being utilized. Additionally, the resources of a single board could be segmented to function as multiple injection sources or multiple systems could function in conjunction to enable a variety of injection and reception scenarios. A loopback arrangement, while useful for test and evaluation, would be unlikely in most applications which would desire this style of network.

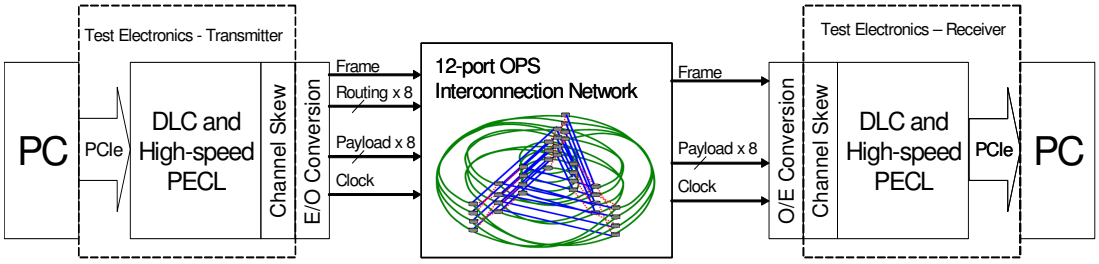


Figure 3.8 System data flow.

One of the diagnostic features highlighted in the figure is the ability to adjust performance parameters (such as channel-to-channel skew) at both the transmitters and receivers on a channel by channel basis to assist in the analysis of system performance metrics. Timing accuracy, in particular, is a major concern since signal skew must be controlled on a picosecond scale to ensure proper alignment and reception of the data across all the payload channels.

The photograph below (Figure 3.9) shows some of the individual components the required to implement this system. In the center of the image is the test development platform with 2.5 Gbps transmitter and receiver modules populated. The board is rotated around such that the 'top' of the board containing the frame and routing signals is facing the left side of the image. Behind the test platform are some of the components for the electrical to optical (E/O) conversion including directly modulated distributed-feedback lasers for the frame and routing bits. Higher-performance payload signals are converted using lithium niobate ($LiNbO_3$) optical modulators for indirect modulation of continuous wave laser sources (not shown).

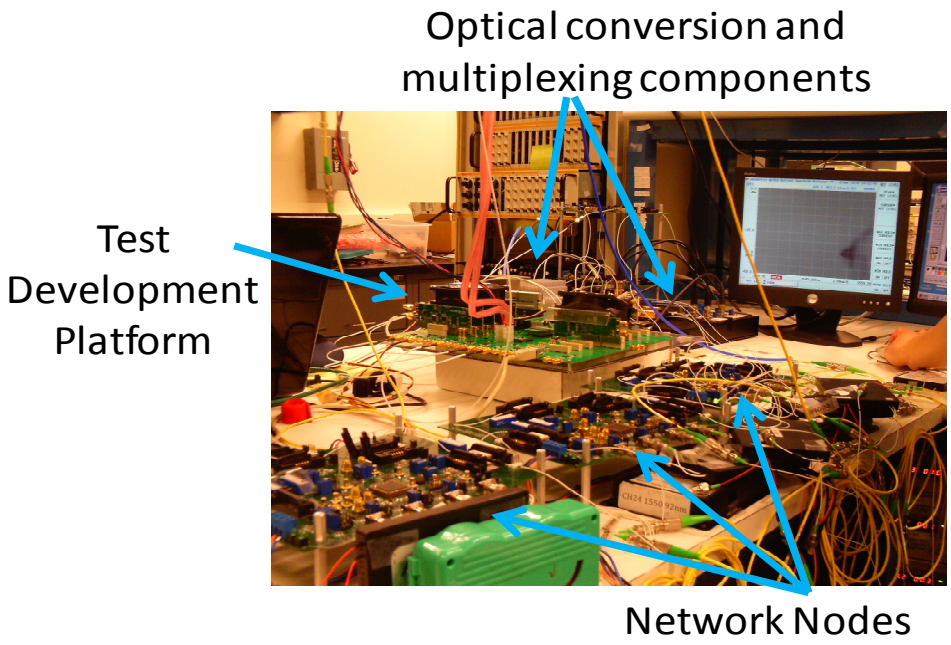


Figure 3.9 System components.

Received signals pass through the bandpass filter shown on the right and the separate wavelengths are processed independently through the O/E conversion board (bottom right). The O/E conversion is performed by 2 GHz p-i-n-TIA receiver modules and passed through PECL buffers to ensure compatible levels with the message processor. While the frame, header, and payload information is preserved all the way to the destination, only the frame and payload signals are passed to the message processor and

used for signal decoding. To avoid the overhead and decoding delay of embedded clock and data recovery, one of the payload channels within the packet is used to clock the received data. This distinction between channel formats is only made at the destination message processor. Within the data Vortex itself, all payload wavelengths are treated equally, i.e. transparently.

Tunable electrical delay circuitry is present on both the outgoing and incoming signals to allow for independent alignment of the signals to compensate for variations in the electrical circuitry and the timing differences in the two modulation techniques. This tunable delay also is essential to facilitate the other design intent of the system, which is the testing and characterization of the data vortex as well as the E/O and O/E conversion techniques being used. The message processor can compensate on a coarse scale for the timing differences between the frame/headers and payload signals operating at different speeds and modulation/demodulation methods. The delay circuitry provides for a finer scale of control, down to a step size of 10 ps, to ensure as close as possible the alignment of the packet at the point of injection into the data vortex. This helps expose one of the largest difficulties in implementing wavelength dense optical switching networks: chromatic dispersion.

3.2 Testing

One of the core features of this network is the payload transparency. The system does not and is physically incapable of altering the signals once they have been injected into the system as multiplexed optical waveforms. This prohibits the incorporation of many test features such as DFT or BIST elements. All manipulation of the signals must occur outside of the system boundary which implies that structural or functional testing must be employed. While there are some structural tests that may be useful to evaluate the electrical control signals and switching logic that exist between nodes, most of the system parameters relate to timing sensitive full-rate signals as they relate to the

operational mode of the system as a whole. For this reason, functional testing is the only option that remains for a comprehensive evaluation of the network.

There are a few system behaviors which increase the challenges associated with the use and testing of the system. The most obvious of these challenges is the bursty nature of the signal transmissions. Due to the packetized nature of the signals, there are bursts of high-speed, very short duration collections of signals. Payload elements are present, at best assuming a constant stream of packets between a single source and destination, roughly half the time, though this arrangement is unrealistic and may dominate the network at the expense of other nodes. Due to the bursty nature of the signals, conventional data capture and deserialization methods such as CDR are impractical. To resolve this issue, a dedicated clock signal is transmitted in parallel with the data as part of the payload to be used at the destination or real-time deserialization without the need to phase lock a receiver clock to the incoming data stream. Unfortunately, this solution is not without its own drawbacks, as will be shown in Chapter 6.

Another concern with the system is that while packet injection is synchronous across injection ports, arrival must be considered asynchronous. Collision prevention within the network will cause the packets to circulate an unknown amount of times and is dependent on the depending on the overall system load. Segment propagation length should be theoretically identical, but minor variations in fiber length and component performance causes slight variations that will accumulate linearly with the number of hops taken by a packet. Collision avoidance may also result in packets arriving out of order relative to the injection sequence.

This combination of test generation (high-speed, multi-channel, and packet formatting), capture (bursty, source-synchronous, and asynchronously arriving), and processing (out of order packets) challenges results in a targeted problem with no solution within reasonable cost and performance constraints given the currently available

test systems and equipment. The driving goal of this research was to develop a test solution that met these specific needs economically while being flexible and adaptable enough to be usable for future and unforeseen test needs of the Data Vortex and different systems alike.

3.2.1 Previous Testing

A concern with modern systems is the ability to fully characterize and test newer and faster designs, especially those that go faster than most test solutions available. This is especially true with optical systems, which can multiplex many independent channels into a much higher aggregate capacity signal. For instance, Figure 3.10 [60] shows a testing configuration used to demonstrate the capacity to transport payloads of 8 wavelengths, each at 10 Gbps. Unfortunately, this sort of testing configuration is unable to verify the complete parallel signal simultaneously, selectively filtering and testing only as many payload wavelengths at a time as are supported by the bit error rate tester.

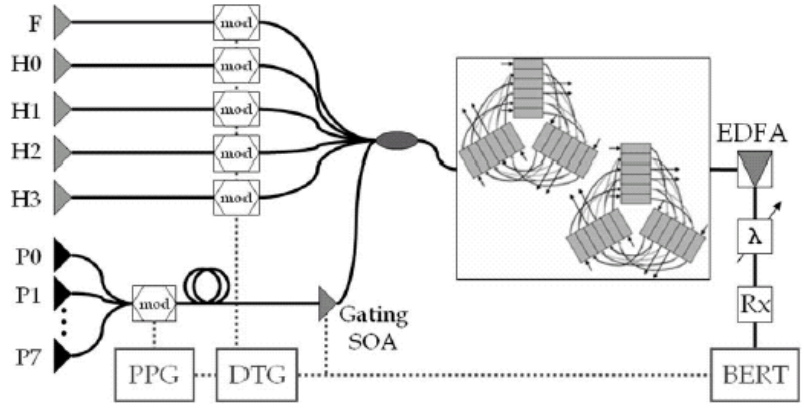


Figure 3.10 Demonstration setup for a 12 port optical packet switching fabric.

This solution also requires many individual pieces of test equipment which must be carefully interconnected and configured to achieve the desired timing and results. In the figure, the box labeled PPG is a 10 GHz pulse pattern generator, the box labeled DTG

is a digital timing generator used to gate the respective packets and a reference clock to the bit error rate tester labeled BERT. This particular configuration allows for the evaluation of a single payload channel at a time, selected using a tunable wavelength filter, over a single propagation path corresponding to a single injection and egress node. A different path will have a different timing relationship and require a resynchronization of the BERT.

3.2.2 Functional Evaluation

Instead of utilizing the various test instruments to create a signal representative of a network packet, the system shown in Figure 3.11 [62] creates and captures an actual packet with no external synchronization between destination and receiver required. The experiment was conducted utilizing a board created as the very first generation of this work, but the fundamental methodology is consistent with the most recent approaches. The test system, labeled as the "Message Processor" and supported by serializer, deserializer, and delay elements in the image, is responsible for generation and formatting of the respective packet signals, including the payload data, frame, routing bits, and the clock utilized at the destination to capture and recover the packet. The same test board was utilized as the transmitter and receiver but no information, other than that required to verify received data against the transmitted data, crosses the system boundaries. The receiver operates on a totally different clocking domain, derived directly from the clock contained within a received packet, from the transmitter logic. This allows the exact same physical test configuration to be used to evaluate more than one propagation path between a single injection and egress node pair without any additional adjustments.

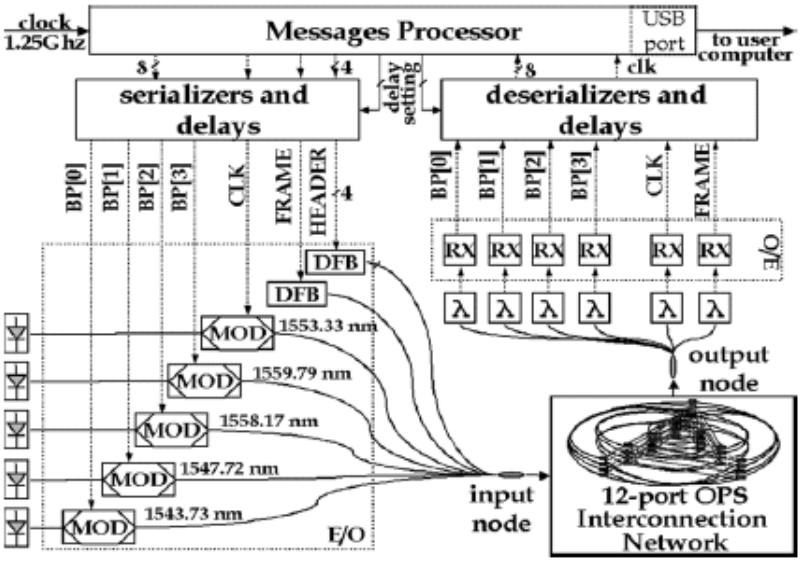


Figure 3.11 Packet generation, modulation, demodulation, and recovery scheme.

3.2.3 Characterization

There are several signal properties that can be varied to assist in the characterization of system performance. Some of these include packet structure with respect to the absolute timing of related signals within the packet, such as the data channels and the parallel clock, and the system response to variations in signal levels or average power.

An effect that could limit the system performance was identified early which is chromatic dispersion. This is the effect that different component wavelengths of a multiplexed signal will travel at slightly different velocities through the transmission medium. In a source-synchronous system relying on clock and data recovery techniques to extract clock timing information from the stream itself, slight variations between the payload channels could be compensated for by using independent CDR circuits for each payload channel. Unfortunately, due to the extremely short signal bursts due to the short packet lengths, CDR techniques cannot be used which is why a parallel clock signal has been introduced as part of the packet payload. However, if chromatic dispersion causes a

timing mismatch to gradually occur across the channels relative to that clock, the overall system performance will be reduced. The amount of this effect and the resulting impact on the capture capability of the payload signals at the destination was one of the first features of the network characterized with this test solution and is detailed in Chapter 7.

CHAPTER 4

TEST DEVELOPMENT PLATFORM

This chapter describes a general overview of the physical design and support capabilities of the test development platform. The system is based around a Xilinx Virtex5 FPGA and uses a combination of fixed support circuitry for generalized applications and plug-in modules for specific and focused applications. In conjunction with application modules for transmission and reception, presented in Chapters 5 and 6 respectively, the overall testing capabilities of this system can be tuned to the specific testing needs at hand. This flexible capability is demonstrated through the implementation of a complete test solution for the Data Vortex, presented in the next four chapters, and a range of experimental modules presented in Chapter 8 which can be adapted or expanded for further testing applications.

The development platform is capable of generating a combination of medium-speed signals, directly driven by the system core FPGA, and high-speed channels custom tailored to the test application. This is achieved through the use of seventeen module plug-in slots. Half of these slots are high-density, supporting 18 slot specific general purpose I/Os from the FPGA and eight are lower-density supporting 10 I/Os. Modules can be interchanged or excluded to adapt the combined system to exactly meet the desired test requirements. Many of the modules discussed as part of this research, especially the 2.5 Gbps serial transmitters and receivers developed for the Data Vortex evaluation, incorporate serialization or deserialization technology to adapt wide busses of slower speed signals to one or more much higher serial data streams. Similar in application to the principle of ATE extension, the data rate and additional characteristics, such as signaling levels or timing accuracy, of these serial streams are improved beyond what would be capable of being generated through application of the FPGA pin I/O

independently. Specifically for the Data Vortex testing applications, the core FPGA generates formatted data passed to the transmitter modules which in turn constructs packetized signal bursts comprised of multiple data and control channels compatible with the Data Vortex. The process is reversed for received packets with the data being evaluated within the FPGA or offloaded to an external system for processing. These modules, signal formats, and test capabilities will be discussed in more detail in subsequent chapters.

4.1 Initial Prototypes

The current test electronics platform is the latest in a line of designs, continually improved to take advantage of newer technologies or to incorporate design techniques learned from previous iterations. Figure 4.1 shows the earliest test system created as part of this research. This design incorporated a Xilinx VirtexII based DLC, visible at the center of the board. Visible at the lower right corner of the DLC is the USB connector for communications with the controlling PC. PECL circuits, responsible for transmission and reception of the signals to and from the DUT, can be seen on the larger, main board. The high-speed output data channels, including the source clock, are to the lower right while the slower speed frame and header signals are at the top. Input logic for the frame and high-speed data/clock is located to the right and the RF clock enters from the bottom. SMA connectors are used to interface differential PECL signals to external electro-optic modules.

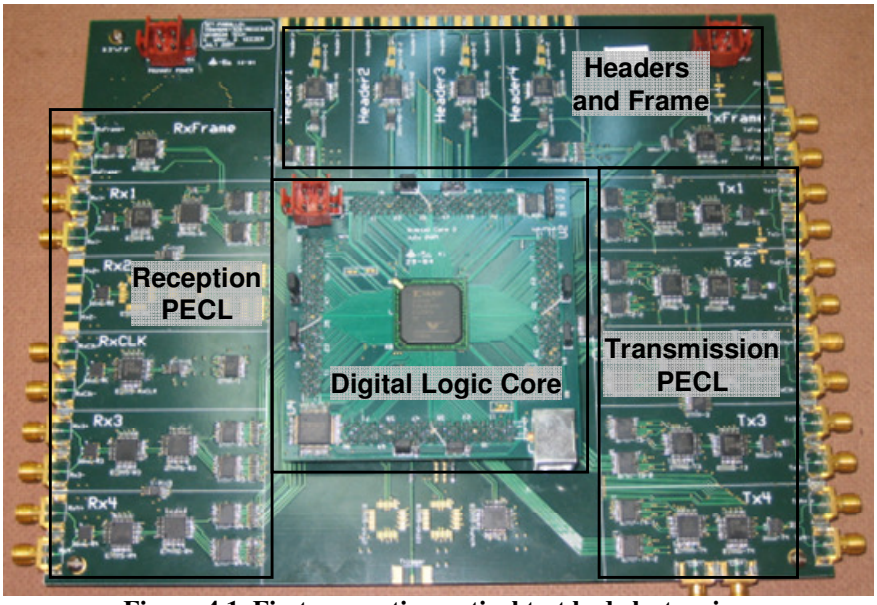


Figure 4.1 First generation optical test bed electronics.

Figure 4.2 shows a cross-section view of the optical test bed electronics. This particular design is an inversion of the current system relationship, incorporating DLC components on a removable module. All of the dedicated serialization and deserialization logic for signal transmission and reception was mounted to a common board. The baseboard also included many of the shared design elements such as clock distribution, delay programming bus, and power distribution network. The DLC mounted in the center of the board via high-speed connectors which are similar to the ones utilized on the current platform for supporting the add-on modules. A cross-section of the design is shown in Figure 4.2 which shows in better detail the modular DLC arrangement. This daughter-card configuration does slightly increase the overall design size and signal latency, but was intended to be offset by the gains in testing flexibility. For instance, multiple DLCs could be kept on hand programmed for a variety of test applications. This also allowed for the possibility of upgrading the FPGA to newer, higher performance cores or adding additional features to the DLC such as external SRAM without a redesign of the PECL systems. Similarly, the PECL circuits could be redesigned while reusing the DLC. Unfortunately, neither of these options were exercised as the design constraints of

using a socketized FPGA as the central element were just too restrictive and the overall quantity of signals generated by the system too low.

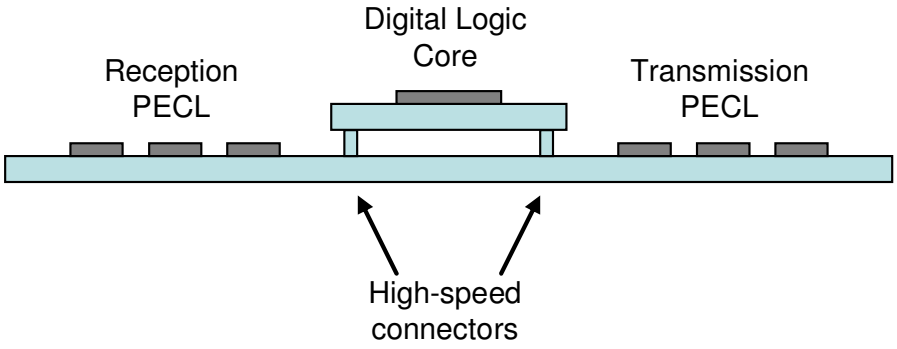


Figure 4.2 Cross-section view of Optical Test Bed electronics.

The need to support additional channels was identified as a significant design requirement and a second design constructed accordingly. The same model of FPGA, a Xilinx VirtexII, was utilized but incorporated onto the board as a fixed element. Without the connectors as a physical bottleneck, the overall design was extended to support double the original quantity of data and routing channels. The revised system supported eight data channels as well as a ninth channel used as a parallel transmitted clock for data recovery at the destination, eight receivers plus one clock, eight routing bits plus frame, and one received frame.

Figure 4.3 shows a picture of the second generation test platform. Structurally, this second version of the test platform is very similar to the current implementation to be presented later in this chapter. Compared to this second version board, the current system relocates the dedicated 2.5 Gbps transmitter and receiver logic, on the right and left sides of the board respectively, onto modular plugin-cards that fit into slots on the main board. This design was the last board implementation that utilized a VirtexII FPGA, but was the first version which included an attempt to incorporate PCI Express connectivity. This was achieved through a single expansion connector wired to a collection of general purpose I/O on the FPGA. A corresponding second module, incorporating a translator

device specifically engineered to interface an FPGA with PCI Express systems, was also designed.

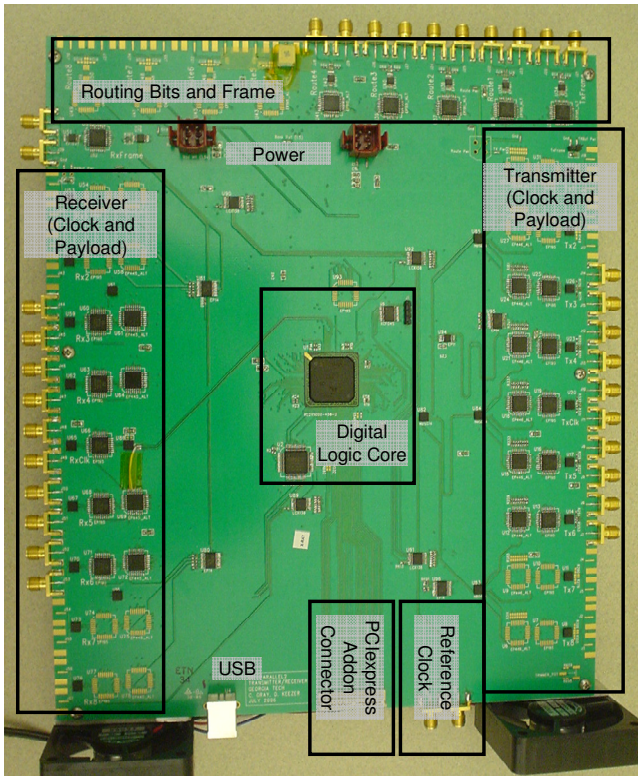


Figure 4.3 Version 2 of the Test Electronics

Both of these early implementations helped identify and explore some of the critical design features required to be compatible with very short-burst source-synchronous signaling. The solution employed, which is still utilized for the 2.5 Gbps modules, is the inclusion of a clock in parallel with transmitted data. The system is still source-synchronous, as the clock travels in parallel with and through the same route as the payload signals. However, the signal can be utilized almost instantaneously at the destination receiver circuitry without the complicated and slow clock and data recovery clock and data recovery (CDR) circuits present in many other source-synchronous systems, especially other optical packet switching network implementations. Additionally, most CDR solutions would be impractical for the extremely short packets

used in this application, requiring 100 or more [54] bit periods to lock to the incoming signal.

4.2 Current Tester Platform

The central processing element of the test platform is a Digital Logic Core comprised of a modern FPGA, I/O and control ports, and high-speed serialization and deserialization hardware connected to the DUT (Figure 4.4). As mentioned in the section above, the earliest designs utilized by this research incorporated a Xilinx VirtexII FPGA. However, the newest design utilizes an updated Virtex5 chip from the same manufacturer to take advantage of newer system features such as RocketIO transceivers and overall improved architecture.

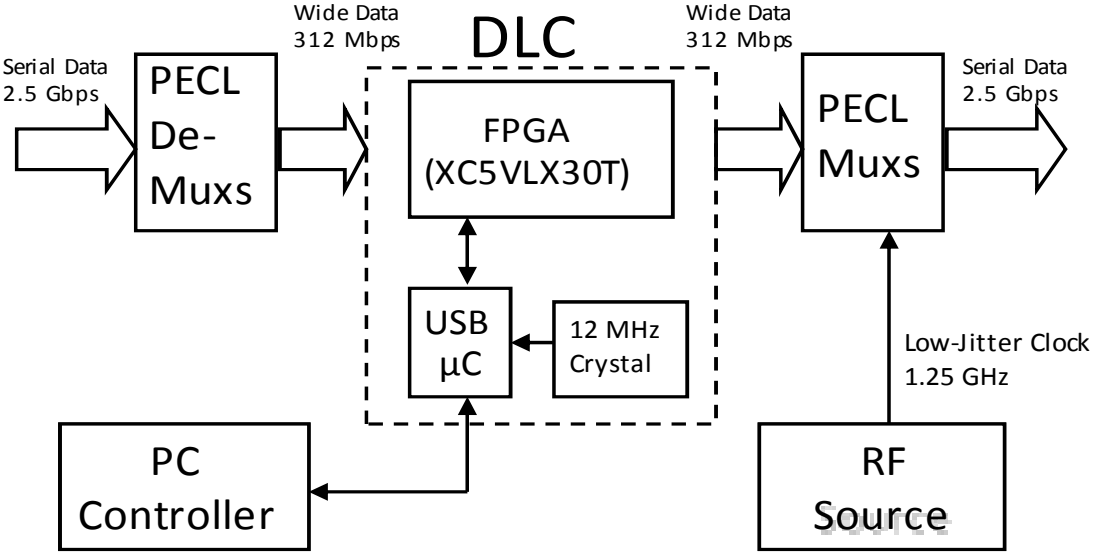


Figure 4.4 Block diagram for source-synchronous test system utilizing Virtex5 based DLC.

External circuitry, indicated in Figure 4.4 as PECL multiplexers and demultiplexers, can be utilized to further augment and enhance the FPGA signaling capability to provide additional functionality such as higher-speed signal support. Older test platform implementations incorporated this logic as circuits on the same physical board as the DLC though the newest design offloads this functionality to removable

modules to enable design reusability and adapt to wider test applications. For a single 2.5 Gbps transmitter channel, as shown in the figure, data is provided from the DLC as an 8 bit parallel bus and processed through an 8-to-1 serializer to the target data rate of 2.5 Gbps. A similar deserialization occurs per receiver channel, converting the high-speed serial stream to a parallel bus at a speed compatible with the FPGA (Figure 4.5). This fixed logic has been adapted for use in the newest design in the form of removable application modules that can be exchanged to adapt the overall test system to alternate target devices to be tested or different data rates.

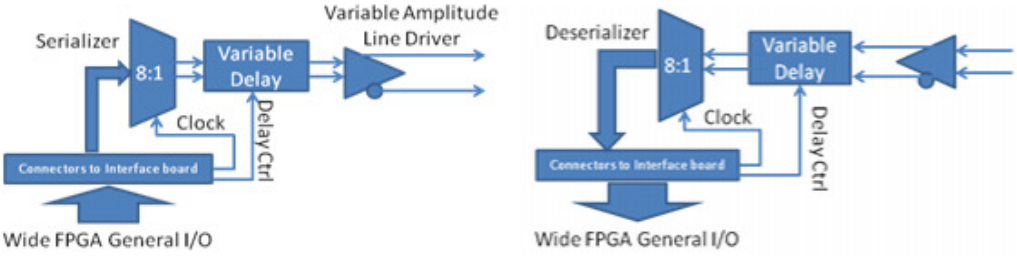


Figure 4.5 Simplified 2.5 Gbps transmitter (left) and receiver (right) module diagrams.

4.2.1 Design Overview

A block diagram for this version of the test platform main board is shown in Figure 4.6. Permanent and fixed elements of the board are shown in blue, including the core FPGA, clock distribution circuits, configuration flash memory, a USB controller chip and a bank of ten fixed I/O channels. Nine of these channels are utilized as transmitters while the tenth is a receiver that fans out to the FPGA and to eight of the reconfigurable module slots (shown as B0-B7). This signal can be utilized to synchronize the entire bank to an external signal such as the framing bit around an incoming data packet. Each of these ten fixed channels as well as the B bank clock input have adjustable output drivers, where applicable, and up-to 10 ns of signal skew capability in 10 ps programmable steps. The programming bus for this signal skew is also available to all of the removable modules.

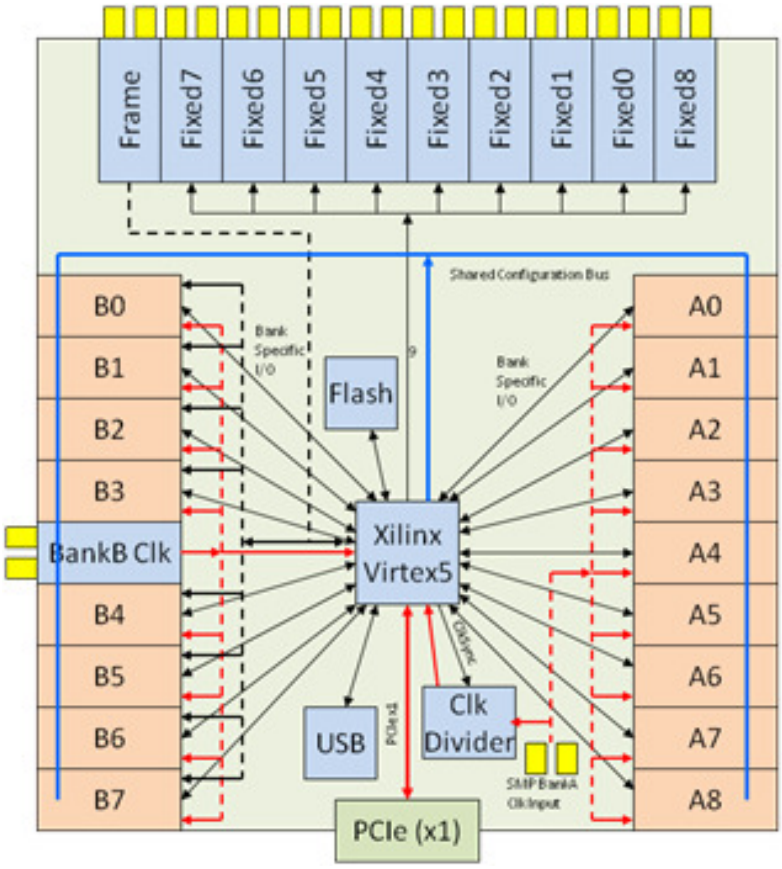


Figure 4.6 Block diagram of the Virtex5 based miniature tester configured for the Bit Parallel application.

There are two banks of configurable module slots to the right (Bank A) and left (Bank B) of the board as shown in orange supporting a total of 17 extension modules. These banks can be independently clocked or slaved to the same external clock source. Each slot is equipped with a pair of 40-pin controlled impedance connectors for general purpose I/O signals connected to the FPGA and a pair of differential SMP connectors for a common clock synchronously distributed across the respective banks. The 40-pin connectors include a variety of signals including dedicated data lines for each channel as well as a collection of common configuration lines shared by all of the slots, such as the delay programming bus described earlier. While Bank A is nominally treated as a transmitter source, the general purpose I/O servicing these slots from the FPGA are configurable as inputs, outputs, or function bidirectionally to support a wide selection of

module arrangements. The same is true of the B bank of module slots, though these are primarily intended for receiver functions due to the presence of an externally driven synchronization or framing signal.

In addition to the USB host interface on this board is a PCI Express card edge connector capable of a single lane of traffic. This connection is processed natively by the FPGA through the combination of an integrated RocketIO SerDes channel and a hardware IP block designed to process PCI Express transactions. This interface allows for very high bandwidth access to the core memory space of the FPGA and can be utilized for any of the control, configuration, and memory operations that can be conducted over the USB interface.

4.2.2 Example Application

One possible application of this system is shown in Figure 4.7 for test and evaluation of the Data Vortex. In this configuration, the processing capability inherent within the FPGA can be utilized for test evaluation of the captured response or the data can be offloaded to the host computer system for validation. The diagram shows a self-test loopback configuration, injecting data into the network and recovering it to be processed by the same test system. A set of serializer and deserializer modules are currently available for population in the A and B banks of this base board to create a test system capable of exercising the Data Vortex at 2.5 Gbps across eight data channels. Though a single test system is shown in the diagram, to perform comprehensive testing of the network would require multiple testers, some of which would be configured to generate additional traffic to load the network and generate potential contentions. Testing the network in this fashion is otherwise unfeasible at this point in time in an economical fashion due to the limitation of the types and quantities of general purpose instrumentation available to the respective research labs.

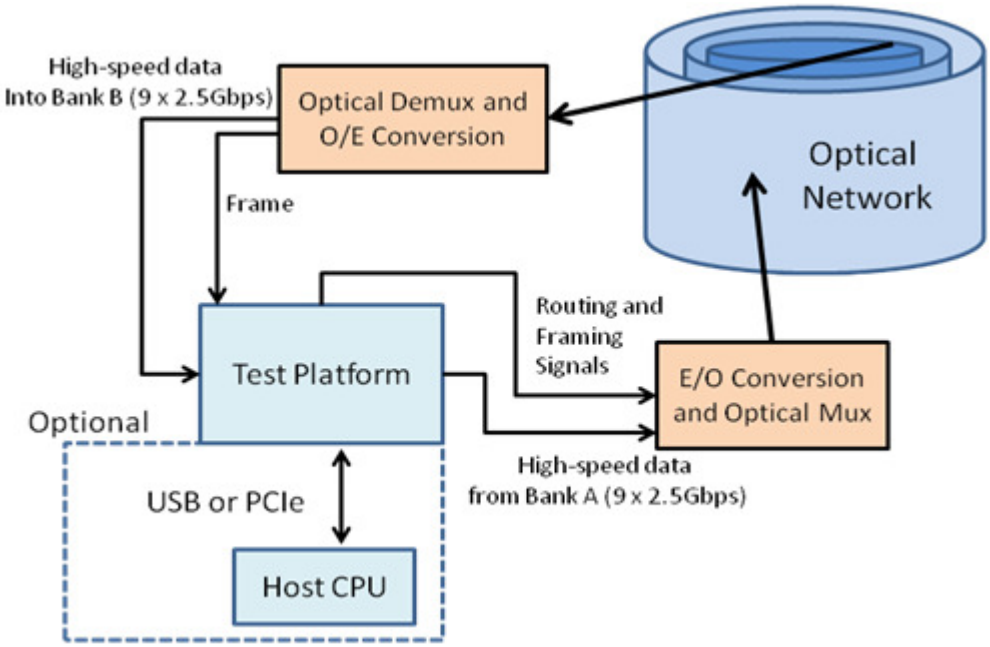


Figure 4.7 System topology configured for closed loop verification.

Additional prototype modules capable of creating a wider range of signal performances, including 5, 10, and 20 Gbps depending on the specific modules and configurations utilized, have been constructed for use with the network or as standalone test elements. These modules and their configuration permutations are discussed in more detail in Chapter 8.

4.3 Plug-in Module Support

While the performance capabilities of FPGA general purpose I/O pins are very respectable, the high-speed, low jitter, and voltage requirements to implement the payload channels for the Data Vortex are outside the device capabilities. To meet the desired signal performance parameters, the FPGA I/O capabilities are enhanced and extended through a set of application modules designed to specifically achieve these goals. While the driving motivation behind the development of this test solution has been evaluation of the primary test bench, the decision to utilize modularized logic instead of

fixed implementations means that the resulting tester architecture can have wide flexibility to support a range of applications, load boards, and experimental designs

The current system is a design evolution over a previous implementation that utilized an older FPGA and hard implemented serialization and deserialization logic as presented in [63]. Many of the base features of the current test system were carried forward from this older system, such as the core DLC functionality, clock and power distribution, and a subset of hard-implemented signaling channels. The overall system has been updated to incorporate one of the latest FPGAs and support infrastructure. The general design approach is to use the FPGA as a centralized control and data source, presenting parallel data to the configurable slots which have application specific logic to serialize and format the serial data to conform to the desired parameters. The reverse process, deserializing the high-speed signals while capturing and processing the parallel data within the FPGA, is performed on the receiving side of the system.

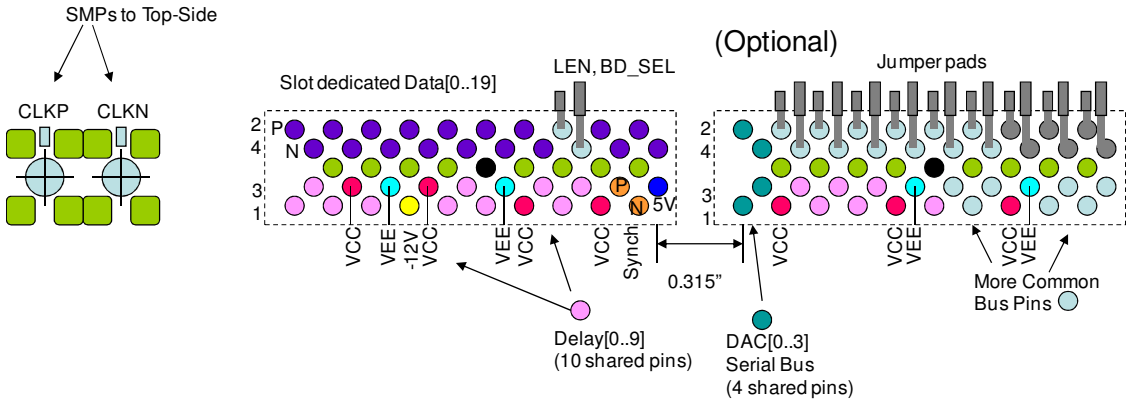


Figure 4.8 Module expansion slot - pin configuration and physical footprint.

Figure 4.8 shows the physical footprint of and some of the signals present on the module connectors. To the left are two SMP connectors which are subminiature style connectors capable of supporting signals up to 40 GHz [64]. These signals, due to the very high connector bandwidth, are reserved for use on the modules as a high-performance reference clock with similar bandwidth capable components supported on the main board. The remaining signals are brought through a pair of 40-pin impedance

controlled connectors. The connector on the left is the primary connector with the majority of the signals required. The one on the left incorporates a number of optional signals that may not be required by all applications. If the signals are not required, the connector can be excluded completely.

Table 2 Plugin-module pin summary.

Description	Quantity	Connector
Reference Clock	1 Differential Pair	SMP
Slot Dedicated Data	18 Single Ended	Main
LEN	1 Single Ended	Main
BD_SEL	1 Single Ended	Main
DELAY	10 Single Ended	Main
DAC	4 Single Ended	Optional
Synchronization	1 Differential Pair	Main
VCC	4/3	Both
VEE	2/2	Both
+5V	1	Main
-12V	1	Main

Some of the transmission modules designed for evaluation of the Data Vortex are shown in Figure 4.9 mounted in the right hand slots of the development platform. These modules are intended for a target operation of 2.5 Gbps each for an aggregate rate of 20 Gbps across eight data channels. Of the 18 available data lines, these modules utilize eight of the slot specific data bits, slot specific delay select LEN, delay programming data, the differential synchronization signal, and high-speed reference clock on the first 40-pin connector and the SMPs. None of the signals included on the second 40-pin connector have been utilized for this particular set of modules. These transmitter and receiver modules are discussed in more detail in Chapters 5 and 6 respectively.

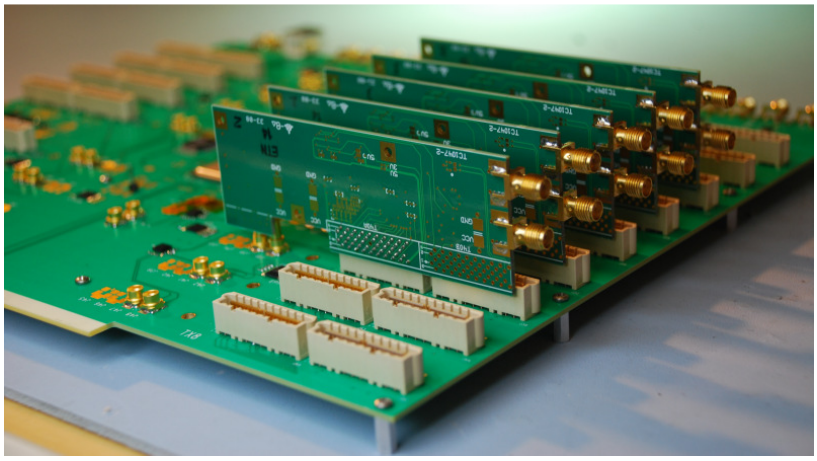


Figure 4.9 Transmission modules for the Bit Parallel application

4.4 Physical Design

Figure 4.10 shows a photograph of the current version of the test platform with some of the prominent features labeled. In support of the primary test bench evaluation, the high-speed serialization and deserialization modules can be slotted into bank slots on the right and left edges of the board respectively. Each side supports eight data channels and a single clock channel. The top edge of the board supports eight routing channels, one outbound frame channel, and one inbound frame channel as non-modularized elements. The incoming frame signal is distributed to all left bank slots and can be used to prime the receiver logic and indicate that a valid packet is inbound to be captured. In the center of the board are the DLC components comprised of a Virtex5 FPGA and supporting flash memory. This particular system incorporates a USB interface and a prototype PCI Express interface through an add-on expansion card (not pictured). Assorted other support circuitry, such as clock distribution trees for the right and left banks, power regulators, and slot enable decoders for the shared bus signals are distributed throughout the main tester board.

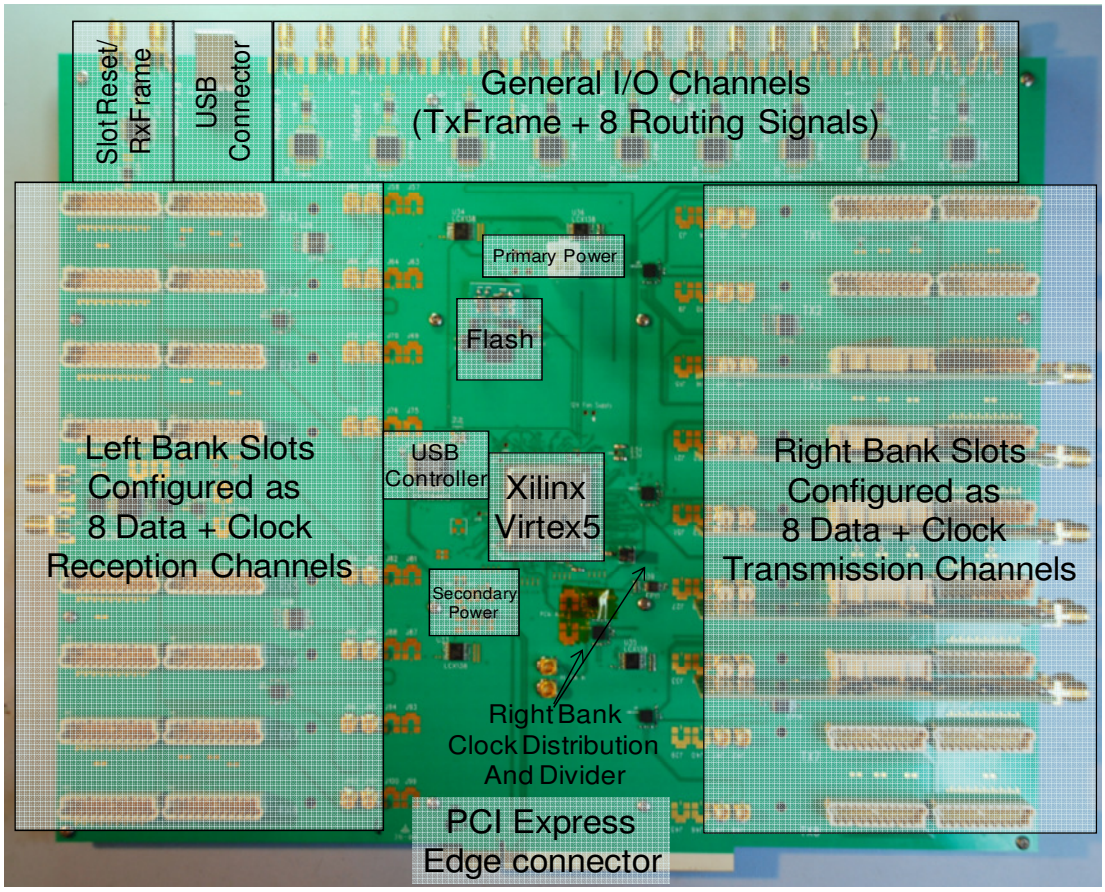


Figure 4.10 Vertex5 based test electronics.

The physical designs constructed as part of this research, including the base platform being discussed here and the modules to be described later, were implemented with high-speed design principles in mind. While it is relatively simple to construct a design intended for very low speeds, at higher speeds many effects such as ringing, crosstalk, noise, and the transmission line behavior of point to point wiring connections become prominent [65]. These effects are not particularly understood by many system designers, especially by older engineers who were educated in older design principles. As a result, larger companies like Xilinx have begun to document some of these effects that should be understood and how a design incorporating their products should be designed to maximize overall system performance [66]. A comprehensive discussion of these effects is beyond the scope of this work, but a basic presentation of some of these

most prominent effects, how particular parts were selected, and how the circuit boards designed for this research were constructed is addressed in this section.

The physical design for the primary test bench was developed logically in the schematic capture component of the Mentor Graphics PCB design software suite PADS. The schematic capture software is called PADS Logic and is integrated with the layout component called PADS Layout. Some of the images below are from the physical layout. Both of these design programs and comprehensive coverage of the design implementation are discussed in more detail in Appendix A.

4.4.1 Impedance

One of the major concerns with a high-speed design is that traces on a printed circuit board (PCB) traces which are utilized as point to point wiring interconnections behave as a transmission line. The trace will have a characteristic impedance which is measured in ohms and is a combination of resistance and reactance. This impedance must be matched with respect to driver strength, line value, and terminations used to absorb signal reflections. The FPGA pins incorporate source impedance matching capability assuming 50 ohm loads, which is a typical target value for trace impedance. The incorporation of termination capabilities within the signal pins eliminates the requirement to add external resistors to match the trace impedance to suppress reflections. This is especially beneficial in an application like this where there are dozens of single-ended lines being sourced by the FPGA and going to the module slots.

The impedance of the PCB traces depends on the geometry of the line itself, the board stackup dimensions, and the layer compositions. The circuit board is constructed from a series of stacked layers of alternating copper sheets and a laminate. The thickness and material of the laminate layers can be varied to achieve an overall desired thickness and alter trace geometries, while the number of copper layers affects the signal layers available for use in the design. Figure 4.11 below shows a ten layer stackup used for

some of the boards created in this research, especially the main tester platform as an overall board thickness of .062 inches or 62 mils is required for compatibility with the PCI Express card edge fingers. Ten layers are available for layout utilizing this stackup, designated as 4 signal layers, 4 ground planes, and 2 power planes. Multiple power planes are necessary due to a number of different power supply values being utilized throughout the board, including 4 for the FPGA alone. 4 ground planes are utilized to provide appropriate reference planes to create controlled impedance transmission lines for the high-speed logic on the board. Two different types of laminates were used in this stackup, FR4 which is epoxy and fiberglass and RO4350 which is a hydrocarbon ceramic substrate with very low loss characteristics ideal for high-frequency signals.

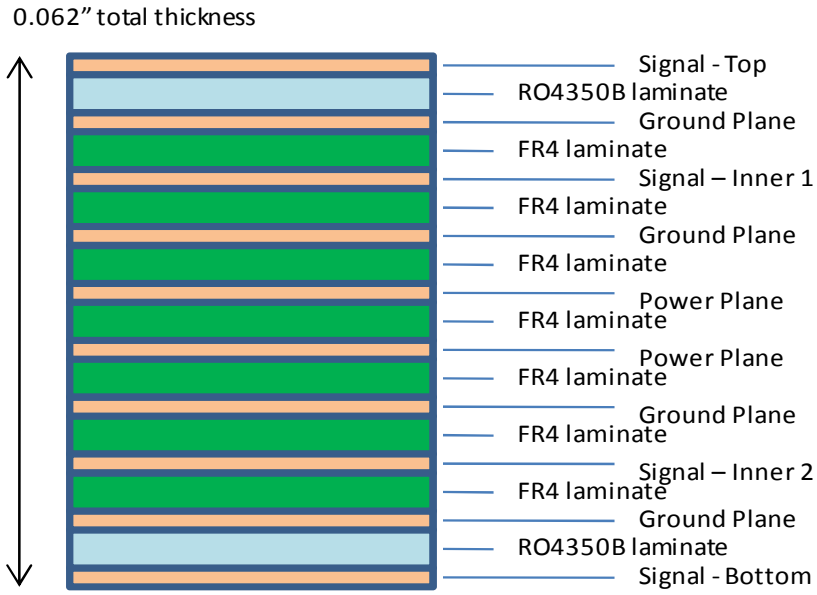


Figure 4.11 PCB 10 layer stackup.

Figure 4.12 shows the geometry of two types of transmission line, microstrip and embedded stripline. The stripline geometry applies to the signal layers buried in the design and surrounded on either side by ground planes, labeled in the above figure as Inner 1 and Inner 2. The microstrip geometry for this stackup corresponds to the traces on the remaining signal layers, labeled Top and Bottom.

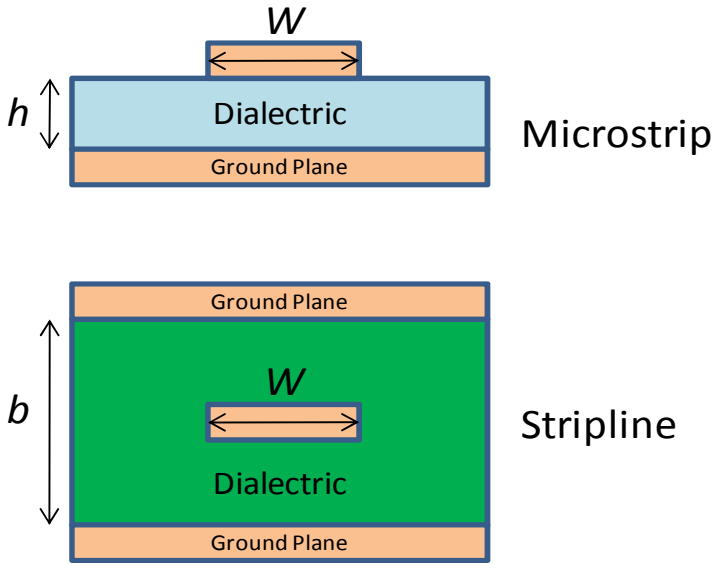


Figure 4.12 Microstrip and stripline transmission line geometries.

The characteristic impedance of a microstrip trace, which includes all of the signal traces on the Top and Bottom layers, can be approximated as

$$\left[\frac{87}{\sqrt{\varepsilon_r + 1.41}}\right] * \ln\left[\frac{5.98 * h}{0.8w + t}\right] \tag{4.1}$$

where ε_r is the dielectric constant of the laminate layer between the trace and the reference ground plane. The width of the trace is w and the height of the trace above the ground plane is h.

The calculation for a stripline trace, which includes the two Inner layers, is similar, with the characteristic impedance approximated as

$$\left[\frac{60}{\sqrt{\varepsilon_r}}\right] * \ln\left[\frac{1.9 * b}{0.8w + t}\right] \tag{4.1}$$

where ε_r is again the dielectric constant of the laminate layer separating the trace from the reference ground planes. The width of the trace remains w. The height value for this geometry is replaced by the parameter b which is the separation between ground planes,

assuming a symmetric trace separation between the planes. Trace thickness t is dependent on the thickness of copper used to plate the signal layers.

Typically, as a part of the board design and creation process, the board stackup will be determined in conjunction with the manufacturer that will be used to construct the board. Based on the customer's preferences, parameters such as the dielectric materials, total layers and order, overall board thickness, and desired trace impedance for a particular trace width can be defined. Of the parameters controlling the value of the trace impedance, trace thickness, plane separation, and dielectric constant are dependent on the pre-manufactured sheets used to construct the printed circuit boards. These board characteristics can be adjusted to a limited extent depending on the particular materials available to the manufacturer in an attempt to meet these parameters as close as possible. Once the board design is finalized, the exact width of a trace for a particular impedance can be calculated by the manufacturer and provided to the PCB designer to complete the design.

4.4.2 Skew

The time of propagation of a signal down a transmission line is not instantaneous. In actuality, like the trace impedance, the propagation speed of a signal within a particular trace is a function of the dielectric material used and the style of trace. For a microstrip trace, the speed of signal propagation can be calculated by

$$85 * \sqrt{0.475 * \varepsilon_r + 0.67} \tag{4.3}$$

and similarly the speed of signal propagation in a stripline trace is

$$85 * \sqrt{\varepsilon_r} \tag{4.4}$$

measured in ps per inch. For the board stackup being discussed here, the propagation speed is 180ps/inch for the stripline traces and 130ps/inch as the dielectric constant for the outer layers and inner layers are about 3.5 and 4.5 respectively [67]. Due to this

difference in signal speed, the final signal alignment or the skew of a set of signals will not only depend on the distance routed but the layers on which the signal is routed as well. There are three particular design components in this design in which skew needs to be minimized. The first is anywhere that a timing sensitive signal, especially a clock, is distributed from one point source to multiple destinations. All destinations should receive the signal simultaneously. For example, the distribution of the Bank A reference clock is from an input set of SMP connectors towards the lower right hand side of the and must travel to a total of 10 destinations (9 module slots + the FPGA).

Figure 4.13 shows the final stage of the high-speed clock reference distribution tree for the Bank A module slots. The clock comes into a 1 to 4 fanout buffer from the previous stage, which is implemented in a similar fashion, and is distributed to three destinations which are the right-most two SMP connectors in blocks of four visible at the top, middle, and bottom right of the image. The distance between the fanout buffer and the top destination is the physically longest, as routed, and therefore becomes the default by which the other two routes will be matched. The bottom route is the second longest and is very close in length to the top route. This small length mismatch can be corrected by slightly overshooting the destination and coming back to complete the connection. The center route is significantly shorter and therefore a larger compensation must be made, visible as a much more exaggerated loop going past the destination before coming back around and making connection.

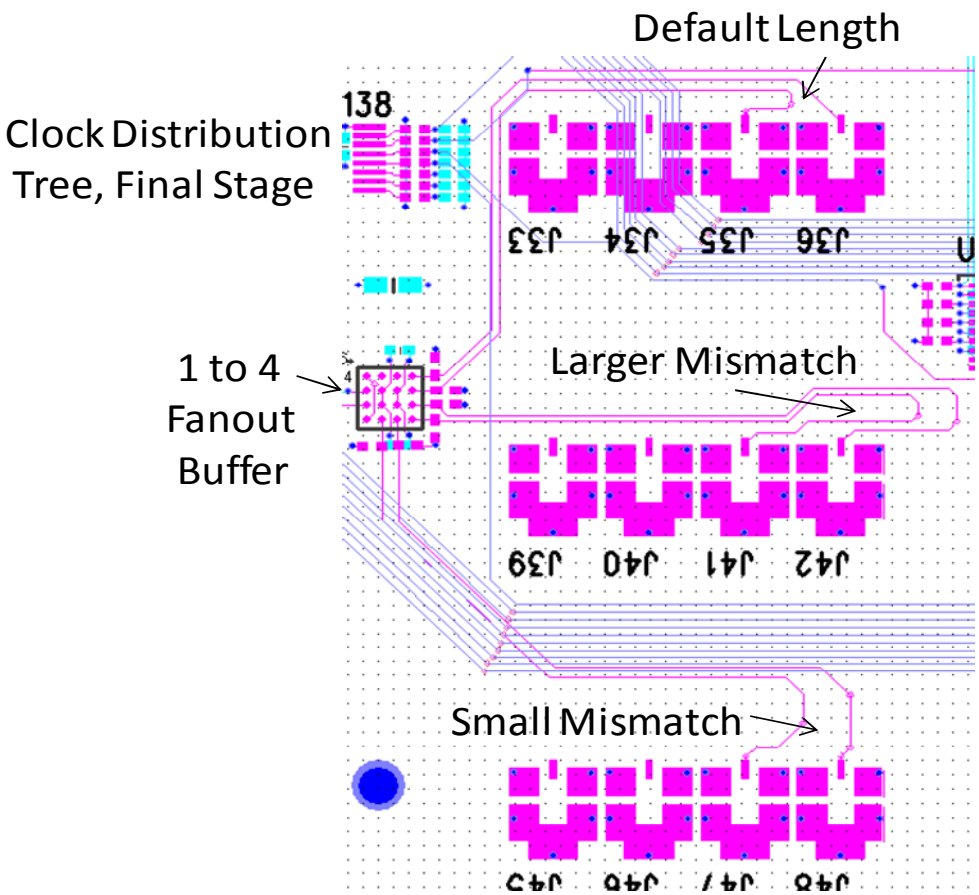


Figure 4.13 Clock distribution tree, final stage.

The second timing sensitive design component is the routing of collections of parallel signals to the modules. The timing skew within the collection must be similar to ensure consistent timing across the entire bus. Figure 4.14 shows one of these bundles of signals leading from the FPGA (off screen to the left) down to one of the Bank A module slots (bottom right, barely visible). In the center of the image only half of the signals appear to be visible because the 18 slot specific data signals are routed on two different layers, Inner 1 and Inner 2, through the same region of the board. The signals break out and are individually visible at the very bottom right of the image. At the center of the image the signals are routed around a standoff mount. This is a hole that penetrates all layers of the board and is used to mount the board to a larger interface, substrate, or to simply elevate the board from a horizontal surface that may interfere with system

operation, such as a metal surfaced table. While there is more than enough space to have routed the bundle to either side of the standoff, it is preferable to keep all of the signals together which allows for a quick visual inspection to reveal that the signals travel a relatively consistent path and should therefore have similar timing skew.

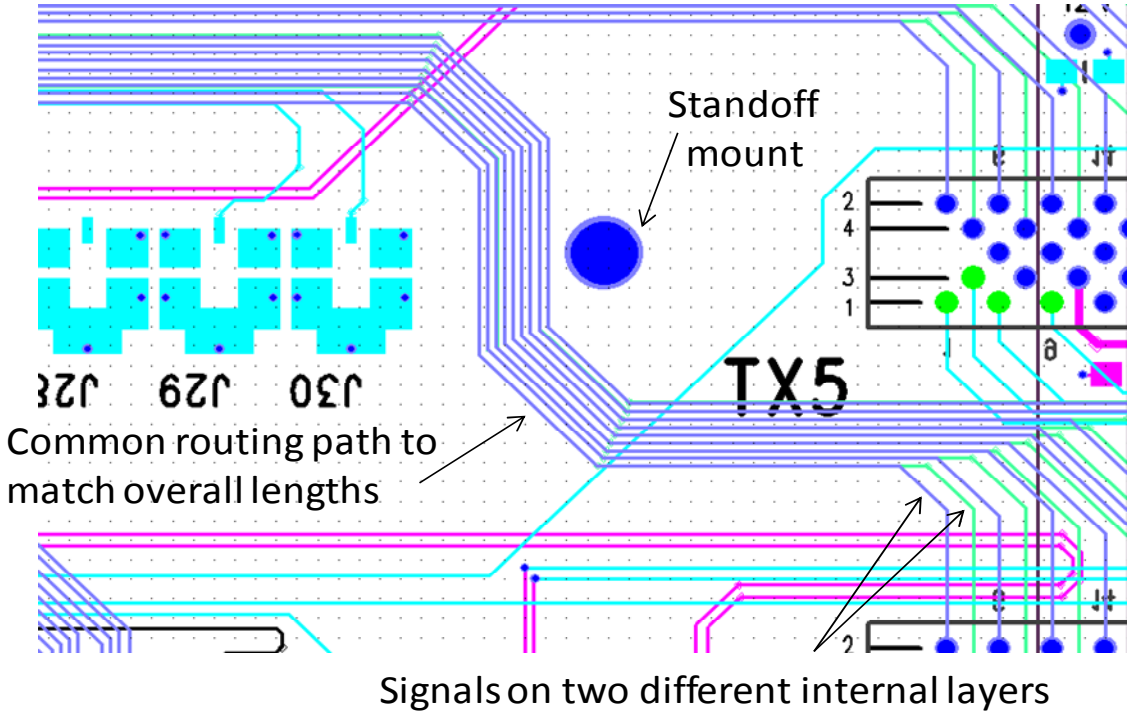


Figure 4.14 FPGA to module parallel data bundle.

The third timing sensitive design component is also related to these parallel data bundles but concerns the bundle-to-bundle relationship. The PCB is a rectangular shape, so some of the inner-most banks are physically closer to the FPGA than the ones further away. In principle, the length of these individual bundles should be identical across all of the destination module slots. However, since this data is coming from or being passed to the FPGA which is running at a much slower overall rate compared to the high-speed serialized signals and reference clock, there is some margin of error. For instance, the trace length of the shortest Bank A signal bundle is associated with the center channel A4 which is often used as TxClk when referring to the Data Vortex testing configuration.

The routed distance from the FPGA to the slot is approximately 3.6 inches, as compared to the slots the farthest away from the FPGA, at the top and bottom of the board, which are nearly 3 inches further away. This distance corresponds to a little more than half a nanosecond. These signals operate at 312.5 Mbps when the high-speed signals are at 2.5 Gbps, so this skew is about 20% of the total bit period of these signals. The FPGA general purpose I/O elements incorporate timing adjustment capability on a pin-by-pin basis which is capable of adjusting the signals slightly in time if this timing margin is insufficient.

4.4.3 Jitter

The highest-speed signals present on the test interface board are the clock references, which also need to be the cleanest signals. Any noise in these clocks will carry forward to any other signals that derive from them, especially the serialized streams. Jitter, small variations from the ideal of signal edges in time, is difficult to get rid of and will carry forward to all subsequent signals. As a result, the best approach is to start with the cleanest source possible. Then any components that are part of that signal path should be as high a quality as reasonably possible and with as few in quantity as possible. For instance, one-to-four fanout devices were used to keep the fanout tree of the clock nets to only two stages deep for a total of 10 destinations. One-to-two fanout devices could have been used but would have required 4 stages of logic and introduced additional jitter. The chips used are differential PECL so it is important to match the two sides of the differential pair in addition to the point to point lengths to ensure the skew timings as above.

4.5 External Control and Data Interfaces

Though the test platform is designed to be relatively autonomous and capable of functioning independently of a larger test infrastructure, interfaces have been implemented for when high level control, configuration, and data transport features are required. The complete interface consists of a memory topology, software interface, and physical interface. The memory topology integrates control functionality, detailed in Chapter 7, and the data regions associated with the application logic. The software interface provides high or low level access, depending on the level of detail required, to these memory and control regions. Two physical interfaces have been incorporated into the current design, including direct access to the Digital Logic Core via a Universal Serial Bus (USB) interface and single lane of PCI Express. However, the design principle is adaptable to any other interface that can be supported directly by the FPGA or adapted externally with an intermediate component as was performed with the USB design presented here.

4.5.1 Memory Topology

To facilitate the support of a wide range of physical plug-in modules, a memory format and addressing system has been implemented as part of the test architecture. This memory system is designed to be a bridge between the application specific logic tied to a particular physical interface and the overall tester design. A memory abstraction layer ensures that design implementations within the FPGA are consistent and reusable across designs. New designs can also be created rapidly and with relative ease as only the application side of the memory interface has to be created while the majority of the control and configuration infrastructure can be reused.

A critical design concern when dealing with source-synchronous systems is the requirement to handle multiple clocking domains [68]. Fortunately, modern FPGAs

incorporate memory elements explicitly compatible with these design requirements. Dual-port memories are designed to operate using two sets of independent data, address, and clock ports, each capable of reading and writing to a single, shared memory block. A diagram of the dual-port memory structures available in the Xilinx Virtex5 FPGA on the test platform is shown in Figure 4.15 [35]. This particular memory element is 36Kb large but can be cascaded with additional memories to support larger sizes. Each port can be individually configured to support data widths of x1, x2, x4, x8 (x9 with parity), x16 (x18 with parity), or x32 (x36 with parity). The particular FPGA on the test system at present, an XC5VLX30T, has 36 of these block memories available for a total of 1296Kb of dedicated internal memory storage [40].

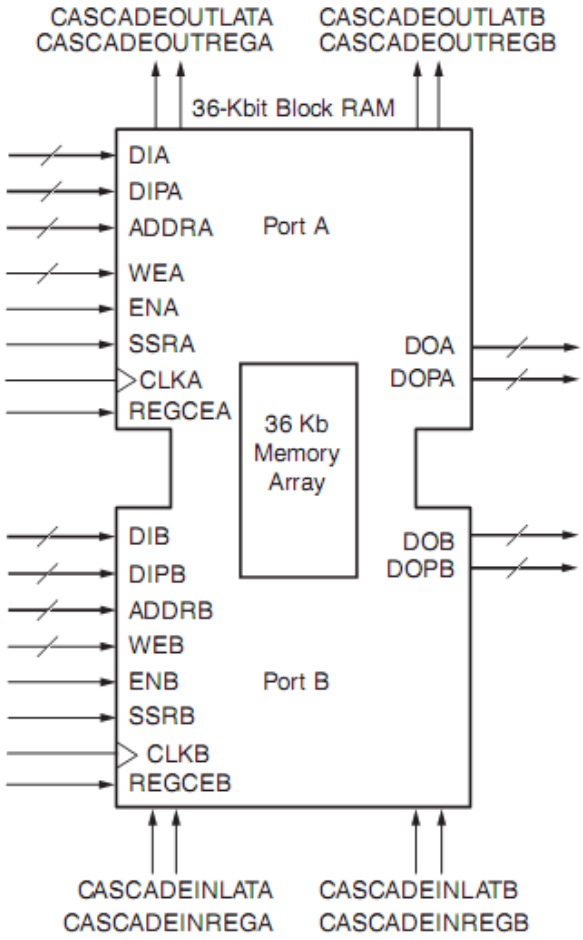


Figure 4.15 Dual-port memory diagram from Xilinx Virtex5.

To accommodate sharing of data across multiple clock domains, which includes the application specific clocks as well as the external interface clock, the dual-port nature of the FPGA internal memory structures can be applied. For instance, a single clock reference can be designated as the system standard and reserved for system wide accesses. All memories implemented within the design that need to be directly accessible externally are connected on a single port with the clock, data, and addressing signals connected to the external interface. This port is also configured for a consistent data width, conformant to the physical interconnect employed.

The second port remains available for the desired application purposes. These roles may vary, such as control, configuration, or test data. An example of this setup is shown in Figure 4.16. This memory element corresponds to a Bank A module, designated in this particular test setup as the TXCLK channel. Port A on the memory is reserved for USB originating data and is 8-bits wide. WRITE_WIDTH_A is set to value 9 because the hardware module actually supports one bit of parity per eight bits of data, but the parity bit is ignored in this application. Port B on the memory is reserved for the application and the clocks driving the respective ports are setup to match. Specifically Port A operates based on SLOWCLK, running at 12 MHz corresponding to the USB interface, and Port B operates based on FASTCLK, running at 312.5 MHz corresponding to the parallel data rate of the 2.5 Gbps transmitter modules.

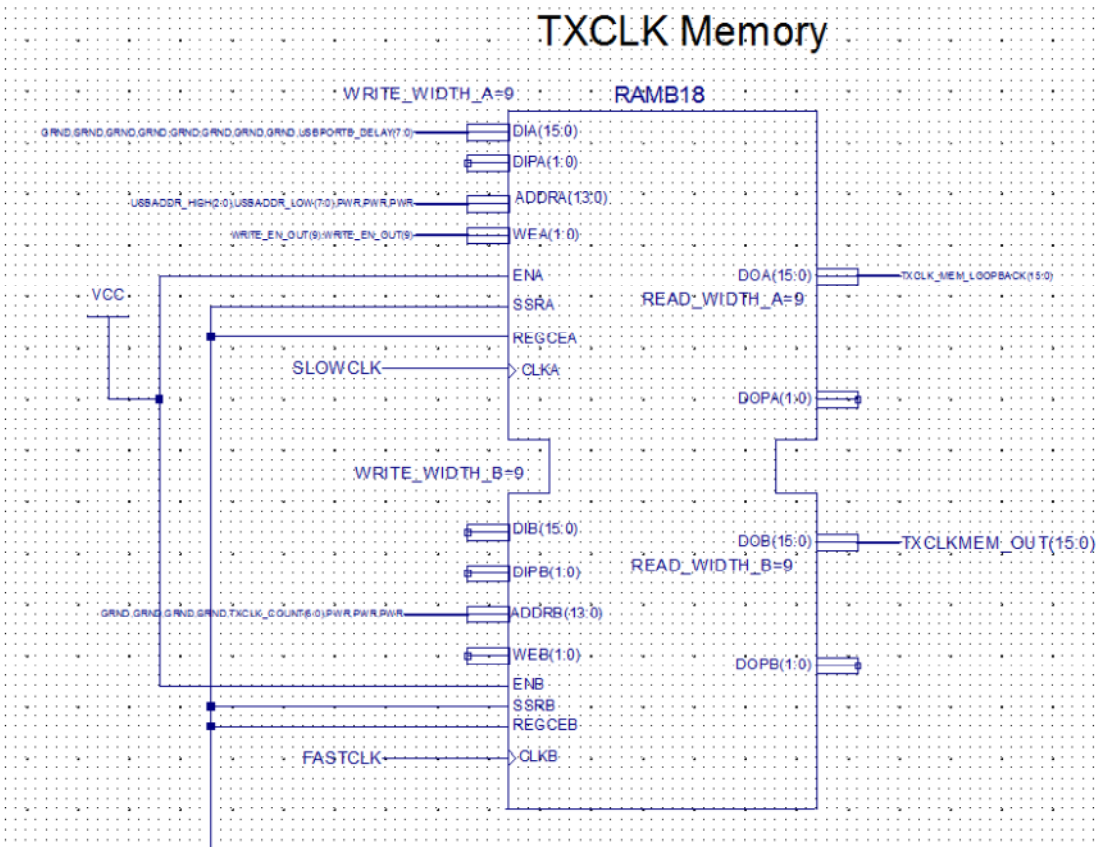


Figure 4.16 Dual-port transmission memory element.

Note that the data input signals on Port B are excluded. The particular test configuration that this image was pulled from does not require that the transmission data values be updated for this particular memory element or any of the other transmitter channels as part of the ongoing test execution. For a more dynamic test configuration, the FPGA logic could be designed to allow for dynamic updates and the corresponding input signals would be connected to one or more data sources.

Utilizing two or more clocking domains creates some design concerns and some care must be maintained to ensure that data crossing these clock domains satisfy the respective timing constraints. Additionally, in the case that two or more system level external interfaces are simultaneously implemented, such as the USB and PCI Express interfaces, one of the interfaces must be designated as the primary and thereby connected

to all accessible memories. A translator module or another memory must be used to bridge the two systems and adapt the memory widths, addressing schemes, and clocking.

4.5.2 Software Interface

The base window of the software interface to the test board is shown in Figure 4.17. This window allows for the control and monitoring of the lowest level unit of data which are 8-bit words using a 16-bit address as configured in high and low paired bytes. This window is generic to the test platform as a whole. Any memory configured in the FPGA conforming to the topology presented earlier can be read or written from this or any subsequent windows. Any additional software modules, accessible from this window, must be customized to the application modules socketed in the system and the corresponding FPGA firmware.

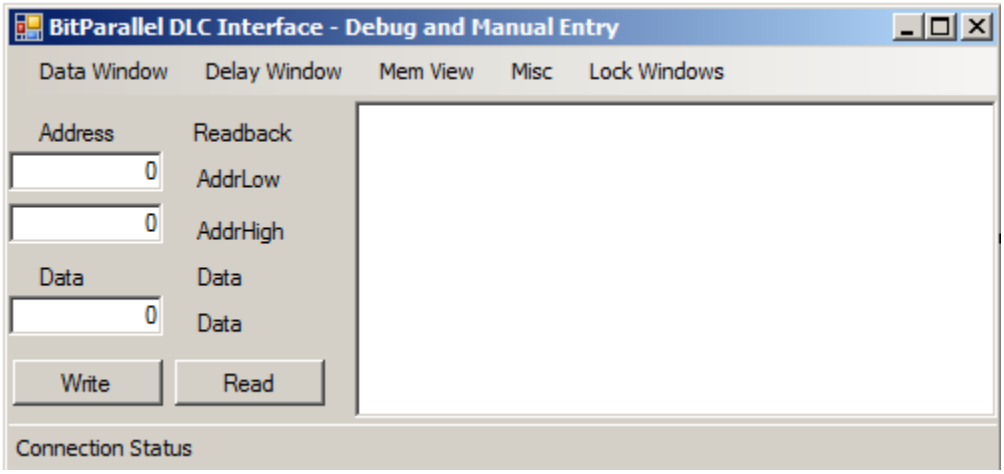


Figure 4.17 Tester interface over USB, base window.

Figure 4.18 shows the first of these modules, customized for the Data Vortex evaluation, which is the delay and configuration window. This interface can be used for adjusting the timing delay of system signals and setting various configuration parameters. All of the plugin-modules and fixed channels incorporate time delay adjustment capability and is discussed in detail in detail in Chapter 5. This software allows for signal channels currently implemented by the test platform and module configuration, labeled

here as a combination of transmitter, receiver, and header/frame channels, to be independently skewed in time by up to 10.24 ns in 10 ps. Many other settings and control features are supported through this window and can be customized to the features implemented by the FPGA programming and specific module capability. Some of the features presented here are the enabling of four of the header signals, enabling the transmitter modules to utilize pseudo-random generated data, and forcing a synchronization pulse to align all transmission modules to a common reference.

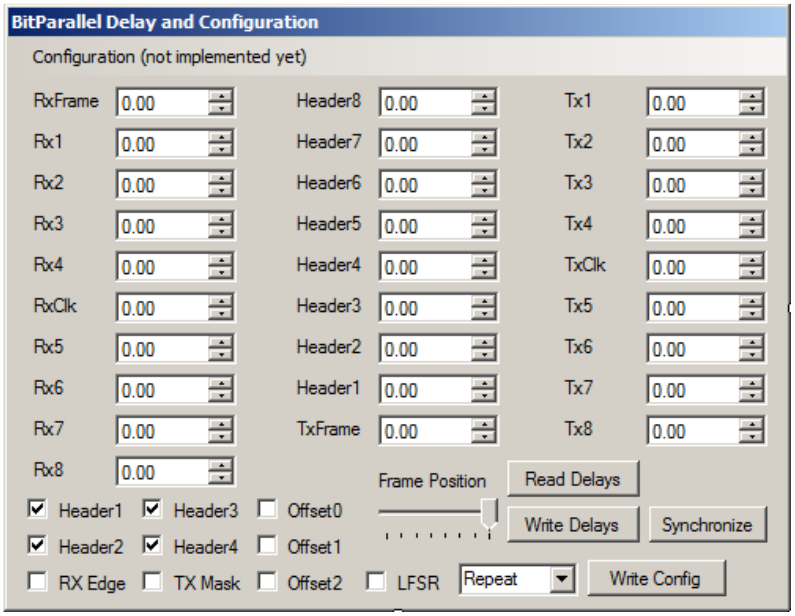


Figure 4.18 Tester interface over USB, timing and configuration window.

Figure 4.19 shows a second window corresponding to the data control and verification functionality. The default configuration utilizes Bank A resources for transmitter channels which are shown and labeled in this image. Eight of the nine channels shown are data while the center most channel corresponds to clock when formatted into a packet. While the clock channel is interpreted uniquely on the receive side of the interface as a clock, for transmission purposes the channel is as modifiable as the other transmitter channels. Data for transmission can be manipulated in the interface as a literal bit sequence and is visually similar to the way the data would appear serialized and observed on an oscilloscope, progressing in time from left to right. In this version of

the interface software, each row is a single signal, with a single packet, or 64 bits divided into 8 bytes per channel, being represented on screen at a time.

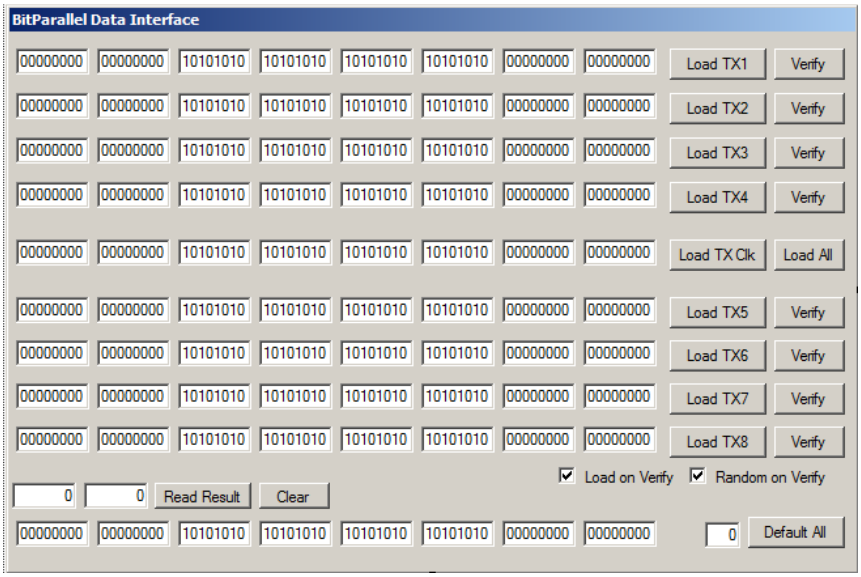


Figure 4.19 Tester interface over USB, data window.

This is an early version of the software which was used to verify the low level operational behavior of the system. Captured data can be observed similarly to the transmitted data in the bottom most row of boxes. A specific receiver channel can be selected by explicit memory address or automatically selected by the software using the verify button to the right side of the transmitter channel data. Checkboxes towards the bottom of the window allow for the software to randomly generate data for the channel per click of the verify button, automatically downloading the newly generated data to the respective transmit buffers and sampling the received data. Implementation details of this software are presented in more detail in Appendix B.

4.5.3 USB Interface

A Universal Serial Bus slave interface is one of the two external interfaces implemented on the test platform. One of the fundamental features of the Digital Logic Core is the incorporation of a USB interface to be used for the issuing of commands,

configuring various settings, and the transport of data sets into and out of the system. The widespread availability of USB on host systems, the ease of programming, and hot-plug capability of the cabling make this interface nearly ideal for the testing application. Unfortunately, even the maximum bandwidth potential of USB is insufficient to keep up with the real time data transport capabilities of one let alone multiple high-speed signal modules. On-chip memory within the FPGA can be used to buffer outgoing and incoming test sequences in small quantities. Real-time processing within the FPGA or sequence compression and decompression techniques, utilized in scan-test applications due to similar bandwidth restrictions, can also be utilized to mitigate these concerns.

The physical layer electrical interface for USB signaling is difficult to directly interface to devices not explicitly designed for such an interconnection, even FPGAs which are inherently flexible. To resolve this issue, a number of commercially available bridge devices have been devised to convert USB signaling into a format more compatible locally to CMOS style devices. One such device is the Cypress CY7C63613 which is a programmable microcontroller interfacing USB signals on one side of the device and CMOS compatible parallel signals on the other. The implementation used in this system is comprised of 16-bit addressing and 8-bit data, which is an improvement over the original implementation of 8-bit addressing and 8-bit data devised by Justin Davis for previous versions of the DLC. Details of this implementation, including microcontroller code for the Cypress device and software for the computer side of the interface are provided in Appendix C. The FPGA portion of the interface implementation is incorporated with the overall design description in Appendix B.

Unfortunately, this particular Cypress device is being phased out. While similar alternatives are available there are design complications which make this particular solution undesirable. The microcontroller code is not persistent across sessions and is wiped when the board is powered down requiring the device to be reprogrammed every session. The process is fast but is still an inconvenience. A bigger limitation is that the

physical interface requires over twenty physical pins to be wired between the FPGA and the microcontroller device. An alternate chip from Silicon Labs, the CP2103 UART bridge chip, is being evaluated as a replacement. While the physical layout differs between the two devices, the host computer programming is different, and the FPGA internal logic must be changed to match, the final memory interface which really matters is identical: 16-bit addressing with 8-bit data. This allows for complete compatibility with the memory abstraction layer and all further system elements are completely reusable. On the host computer side, a properly structured Read/Write function library can ensure that the majority of the interface is similarly reusable.

4.5.4 PCI Express Interface

USB ports are available are on most computers, easily accessible over long and convenient cables, and programming for the interface is relatively easy to implement which is why the majority of the discussion and practical efforts in this research focus on this interface type. However, due to the way the memory addressing scheme is implemented within the FPGA it is simple to implement alternate interfaces as long as the physical PCB layout, FPGA programming, and host software interface are structured to be compatible. PCI Express as a signaling interface brings to the design a number of benefits such as improved signaling bandwidth, but is a relatively difficult challenge to implement physically.

The first system that was intended to incorporate PCI Express support was the second revision of the test platform presented earlier. Due to the physical size and orientation of the signal connectors on the board, the design was incapable of physically fitting within a computer to be slotted directly into a connector slot. Additionally the FPGA being utilized at the time, a VirtexII, was incapable of supporting the 2.5 Gbps signaling required to directly implement the PCI Express connection. To solve both of these problems a small extension board, shown in Figure 4.20, was created incorporating

a physical layer interface chip, similar to the USB interface chips presented last section, which is capable of converting the PCI Express signaling to a parallel interface more compatible with devices such as FPGAs. The interface board is also long and equipped with a connector to interface to the test platform board which can be outside of the host computer case.

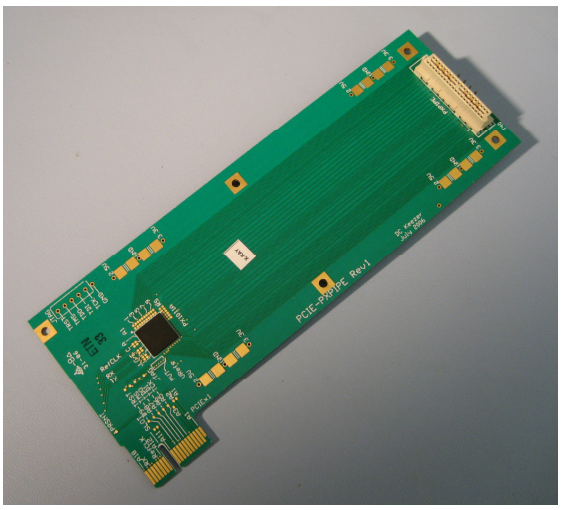


Figure 4.20 PCI Express interface card.

The interface protocol between the PHY chip, a PX1011A manufactured by NXP, is shown in Figure 4.21 and is referred to as PXPIPE [69]. Much like the ATE extension modules discussed earlier and application modules designed under this research, the high-speed PCI Express interface is converted to a higher quantity but lower speed collection of signals compatible with the FPGA. A block of protocol processing logic can be further implemented inside the FPGA to handle this interface and complete an effective connection to the host PC. Unfortunately it was eventually determined that the compilation requirements of this processing logic and the remaining logic to perform all of the test requirements of the system were well in excess of the logic available in the FPGA.

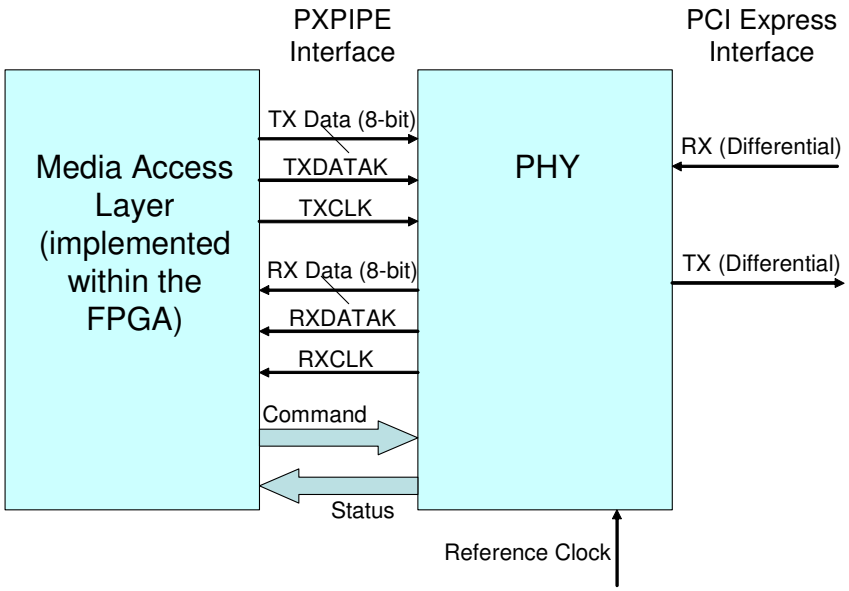


Figure 4.21 PXPIPE signal diagram

This space limitation was one of the guiding motivations to develop the third and current test platform, incorporating a Virtex5 FPGA which provided an opportunity to increase the overall amount of programmable logic available. This model of FPGA also incorporates RocketIO SerDes transceivers which can be combined with a dedicated block of logic capable of processing PCI Express packet traffic, allowing for a direct implementation of PCI Express interface without the requirement for any external devices. The physical constraints still existed with regards to getting the test platform into a computer case though this was solvable through the use of commercially available PCI Express cabling adaptor cards. The cards, utilized in pairs, allow for an oversized PCB such as the test platform to be connected to a host PC over a cable allowing the two systems to be physically separated. Figure 4.21 shows one of these two cards connected to the test platform (top) and the corresponding cable (left).

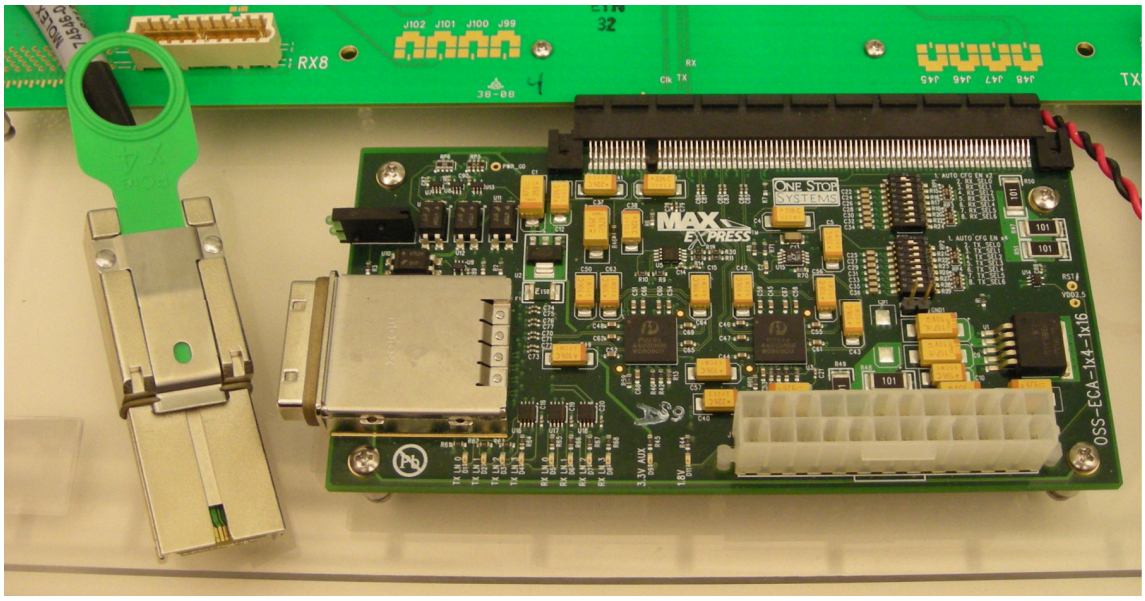


Figure 4.22 Commercially available PCI Express cable adaptor card.

The current PCI Express interface has been tested and verified as functional and limited data transport operations have been completed using the interface. However, the inclusion of additional features is limited by the need to develop a system driver compatible with the board and the development of additional processing logic within the FPGA to react to incoming data appropriately.

CHAPTER 5

TEST SIGNAL GENERATION

As shown in Chapter 4, a number of data and control signals are provided to the various plug-in slots available on the test system main board. Using these connections, a variety of utility modules can be constructed creating a wide array of test signals which are demonstrated here using single channel, 2.5 Gbps transmitter modules designed for Data Vortex testing. These modules demonstrate a combination of application specific features such as 8 to 1 parallel to serial conversion as well as a subset of features available to all modules such as timing and voltage adjustment.

5.1 General Capabilities

A high-level block diagram of the general features implemented by the 2.5 Gbps transmitter modules and lower-speed general channels along the top of the test system board is shown in Figure 5.1. This figure also displays the relationship of these modules with the Digital Logic Core. The low-speed channels and module slots share a common delay programming bus while data, as a single bit or 18-bit parallel bus respectively, is unique to each destination. When the system is configured for a 2.5 Gbps serial signaling rate, the FPGA, and all of these data interfaces correspondingly, operate at 312.5 Mbps. The high-speed transmitter modules incorporate serialization logic that converts this lower speed data in 8-bit parallel words into the higher-speed serial equivalent. All of the transmitted signals pass through delay buffers and variable amplitude drivers, providing the ability to independently shift all channels by up to 10 ns with 10 ps resolution and alter the signal amplitude. This is useful for characterization of the DUT, varying signal alignment and levels to evaluate alternate physical interfaces, and ensuring timing accuracy across the parallel channels.

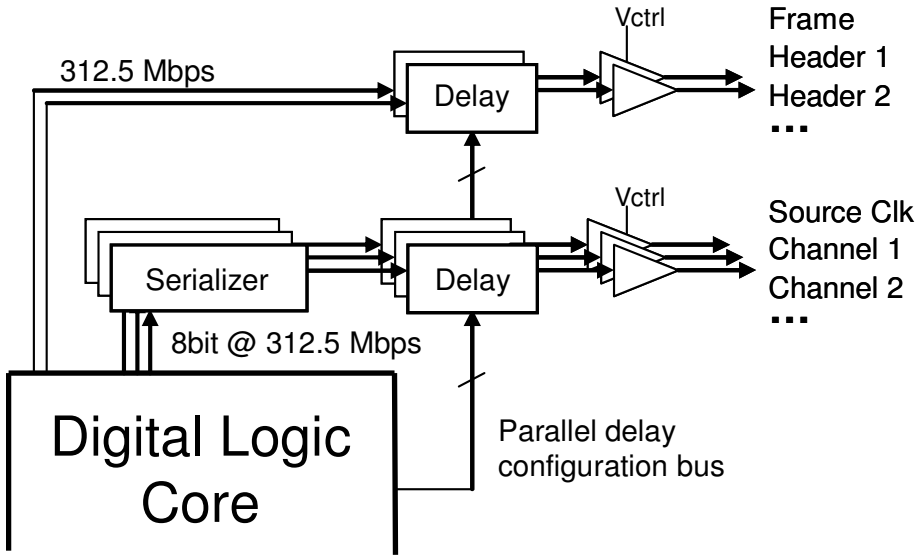


Figure 5.1 Transmission Logic Diagram.

A basic block diagram for the logic implemented on a single 2.5 Gbps transmitter plug-in module is shown in Figure 5.2 8-bit wide single ended data from the FPGA is processed through an 8:1 serializer device utilizing a high-quality reference clock synchronously distributed to all of the system modules. The output of this device is a differential PECL signal that then transitions through a variable delay buffer. The delay amount is programmed on a slot by slot basis allowing all channels to be independently skewed relative to one another. A high-performance, variable amplitude silicon germanium (SiGe) output buffer-driver is the final device element before the transmitted signal exits the board through high-performance, edge-launch SMA connectors. The nine utility slots directly implemented along the top edge of the test board are designed similarly, incorporating identical delay buffers and a similar though slightly lower performance buffer-driver. The serializer logic is excluded from these channels as the data originating from the FPGA is a single bit wide per channel.

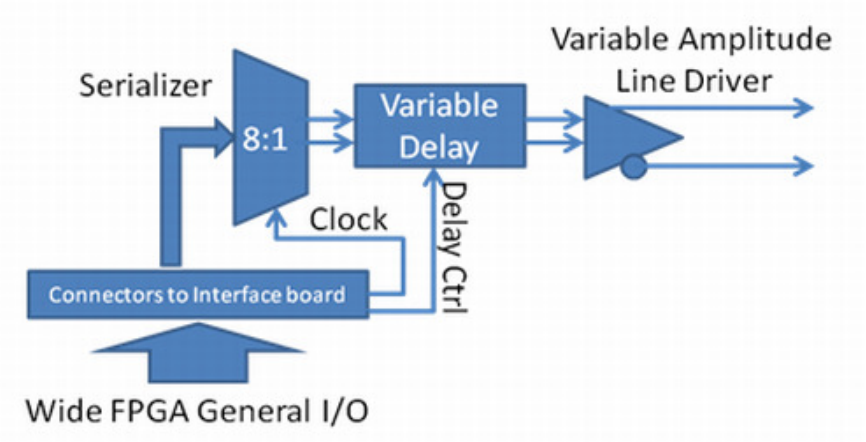


Figure 5.2 2.5 Gbps transmission module logic diagram

Figure 5.3 shows an eye diagram with the output operating at the project target rate of 2.5 Gbps. For this test, the output waveform is a 2^N-1 pseudo-random bit pattern produced by a linear-feedback shift register (LFSR) encoded into the DLC FPGA. In addition to fast rise and fall times, the silicon germanium output buffers introduce very little jitter, which was measured at the crossover point. For this signal, jitter was measured to be 46.7 ps peak-to-peak, resulting in a usable eye opening of 0.88 unit intervals (UI). While these modules were designed for operations at 2.5 Gbps, the test electronics have been used to generate sample signals at 4.0 Gbps with similar results. The measured jitter at the crossover point was 47.2ps p-p with a usable eye opening of 0.81 UI and no visible signal attenuation. This bit rate is at the upper limit of some of the individual PECL components. We independently measured the 20% to 80% rise and fall times on a single edge and found them to be in the range of 70 to 75ps. The jitter on these individual transitions was only 24ps peak-to-peak (about 3.2ps rms). More detailed performance metrics are presented at the end of this chapter.

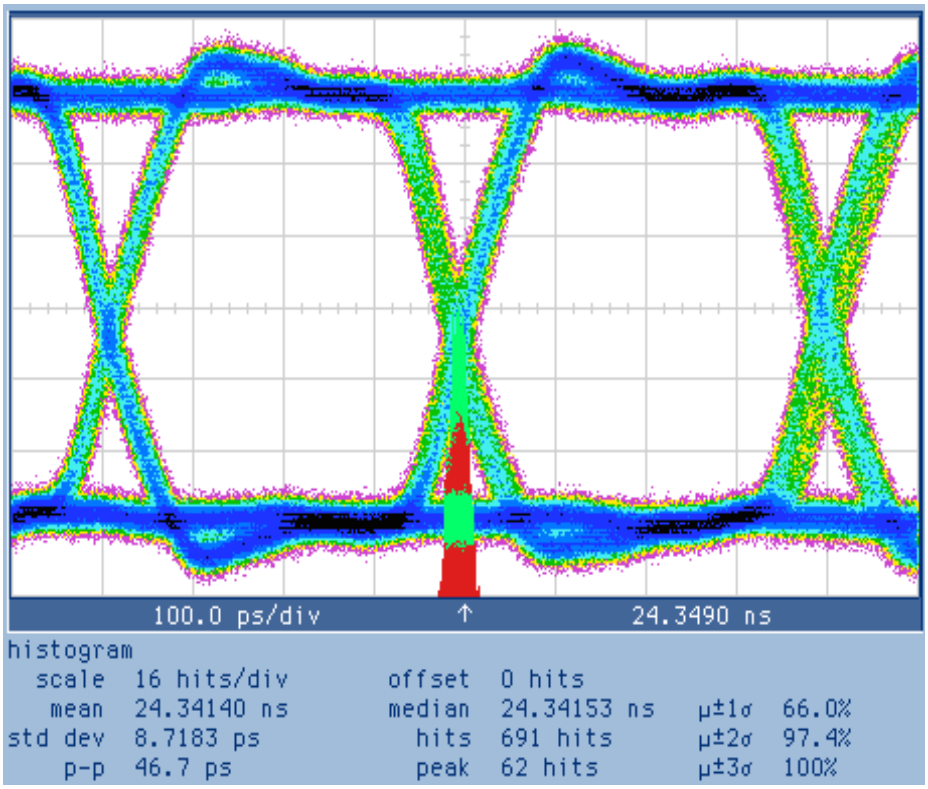


Figure 5.3 Example 2.5 Gbps eye diagram.

5.1.1 Timing Adjustments

Critical to the mult-GHz testing process is the ability to control timing of logic transitions with fine precision, measured in picoseconds. To accomplish this, the various modules implemented in this work and the fixed channels on the test platform include integrated delay generator circuits. These circuits are made up of a series of digitally-controlled elements and an analog-controlled vernier delay element. There are two different ways to achieve a timing adjustment and that is through direct manipulation of the serialized data stream or skewing the reference clock. The direct adjustment method was utilized for many of the implementations in this work. This allows for the most flexibility, enabling the shifting of each channel across the full timing adjustment range of the delay chip relative to all of the other platform signals.

The particular delay circuits used in this work are implemented as a cascade of 10 distinct delay stages, each roughly corresponding to a power of 2 multiple of 10ps and controlled by a 10-bit programming word. The least significant bit corresponds to the first stage which is 10 ps and the most significant bit corresponds to the final stage which is 5120 ps. Activated individually or in combination, the stages can be combined to create a total delay across the range of 0 ps to approximately $10x2^{10}$ ps or 10.24 ns in 10 ps steps. The vernier range is approximately 40ps, and is controlled by the analog voltage outputs from 12-bit DACs.

In actuality, the total usable adjustment range of these delay chips is shorter than the theoretical amount. To achieve the highest timing accuracy possible, the performance of these delay chips were measured and a calibration scheme devised. The actual amount of delay for each of the 10 stages can be measured independently. Theoretically, the total chip delay should correspond to the combination of the activated stages or, conversely, the programming value for a desired delay value can be calculated through the combination of stages required to meet the desired total delay. Once the actual delay values associated with the 10 distinct stages are known then the actual delays for an all the 1024 binary codes can be accurately predicted.

A plot of measured versus predicted delay values are shown in Figure 5.4 for 50 representative timing values spanning the 9.2 ns range in 200 ps increments. Only 9.2 ns versus the theoretical 10.24 ns are utilized in these measurements because the individual stages measured consistently low by about ten percent of expected value. On a full-scale range, the plot appears nearly linear, so a simple line-fit might suffice for predicting the remaining 974 points. However, as shown in Figure 5.5, the residual timing errors (after calibrating the 10 binary stages) are generally about 5 to 10ps, and occasionally as high as 12ps. This uncertainty is acceptable for slower speed signals (<2Gbps where the bit-period is >500ps). However, as the bit-period decreases (to 200ps at 5Gbps), tighter timing accuracy is needed.

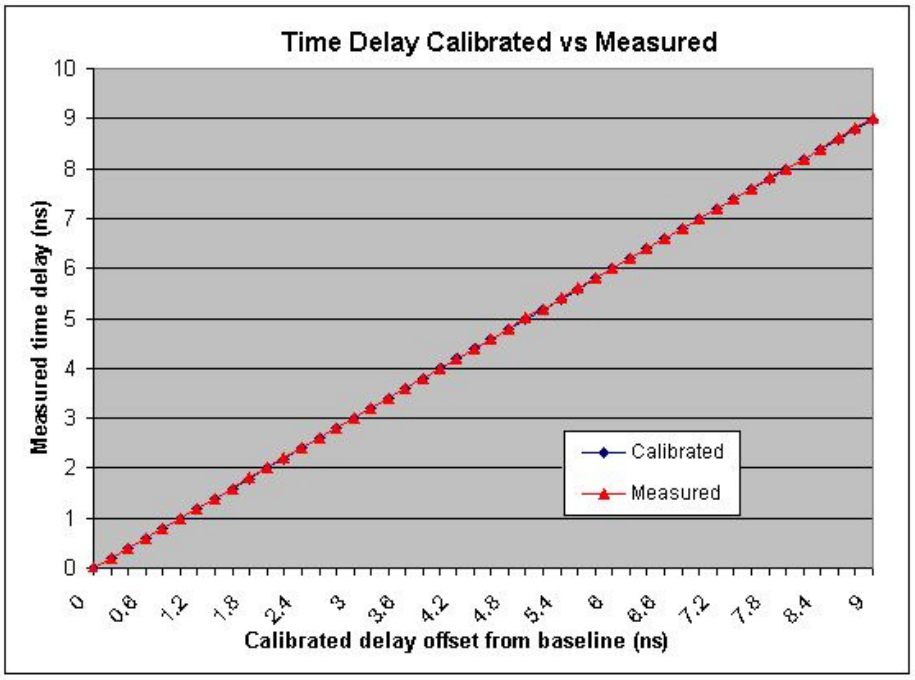


Figure 5.4 Linear programming range for digitally-programmed time delay. 200 ps increments, across a 9.2 ns range.

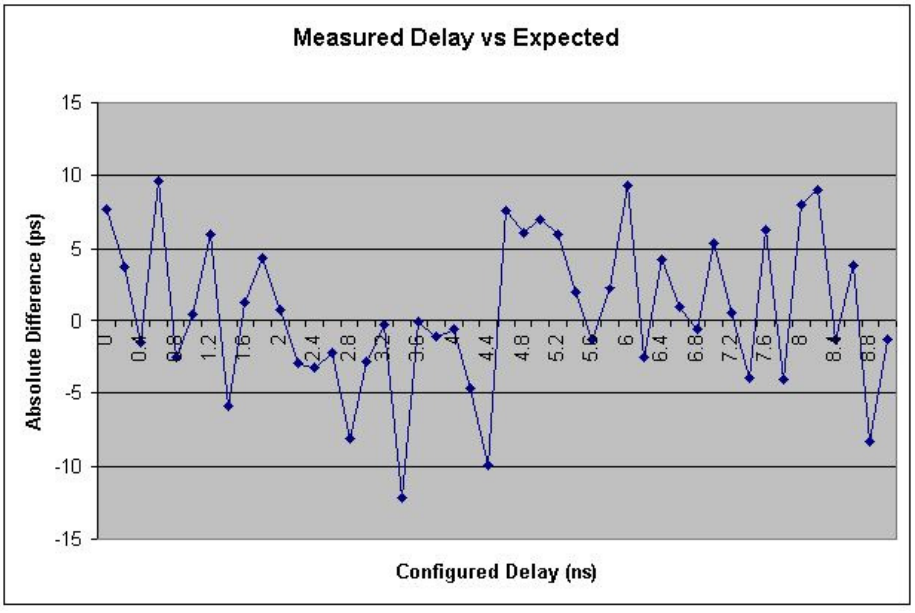


Figure 5.5 Residual timing errors of calibrated delay values vs measured values. Errors are generally <10ps, but occasionally a bit higher.

Tighter timing accuracy is achieved using an analog-controlled vernier circuit. The circuit adjusts the signal delay value through a 40 ps range, using a reference voltage.

To achieve the desired 40 ps delay range, the reference voltage is adjusted through a range of about 1.3V. The modules include a provision to be populated with components capable of adjusting this control voltage in sub-millivolt steps. This can be achieved programmatically using 12-bit DACs controlled from the FPGA or a trimmer potentiometer used as a voltage divider adjusted manually. In principle, sub-picosecond resolution can be achieved with this methodology. Figure 5.6 shows five examples of accurate timing control, using 2ps steps and 0.2ps resolution. The particular signals shown are generated from one of the dual-channel 2.5 Gbps modules which will be presented in Chapter 8. However these signals are are representative of most the other modules that use this same delay circuit. The vertical and horizontal scales are greatly expanded so that only the critical 50% crossing point for a particular rising edge is visible.

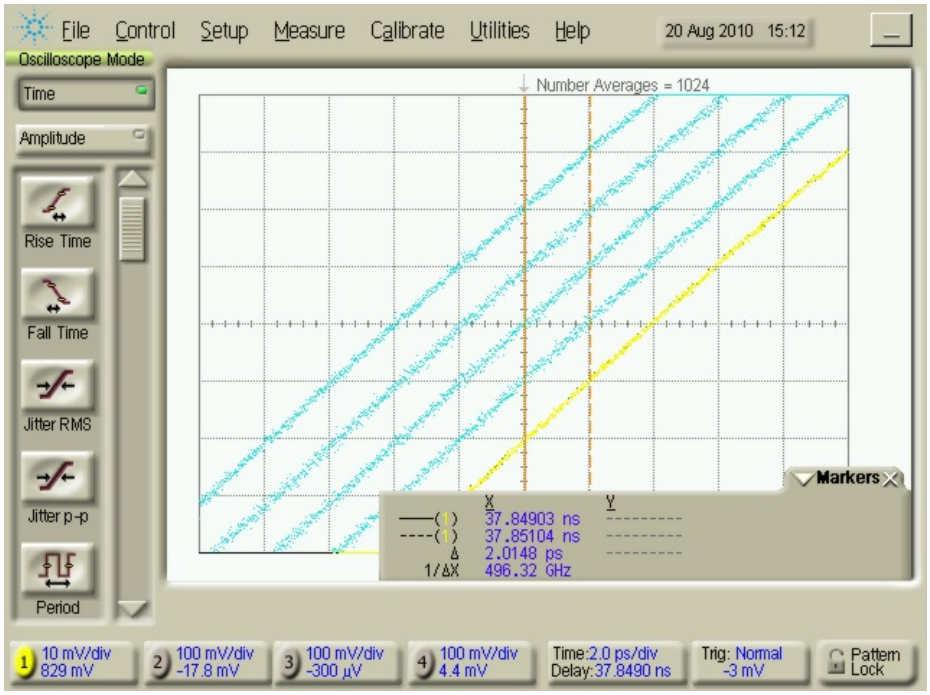


Figure 5.6 Analog tuning delay at 2ps spacing.

The timing vernier circuit is used to correct/adjust the residual timing errors for each delay value after digital calibration is performed. For this to be completely effective, each of the 1024 digital codes must be separately measured and stored in a look-up table.

The vernier range must also be calibrated and characterized because the response is not strictly linear either. A representative calibration of the vernier section is shown in Figure 5.6. This figure shows several measured delay increments as a function of the programmed control voltage. Fortunately, the response is mostly linear throughout the range, as can be seen in Figure 5.7, and therefore only a few calibration points are required.

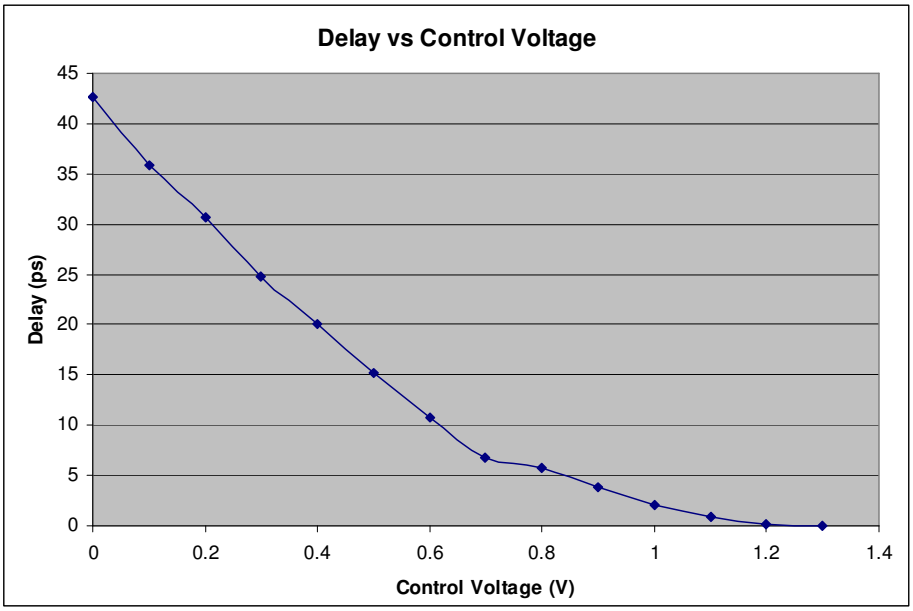


Figure 5.7 Analog delay vs voltage.

With this combined calibration information, accurate programming codes can be generated for a specific desired delay. This adjustment can be performed in the control software or processed through the FPGA firmware as desired.

5.1.2 Data Sequencing

Signal sequences can be retrieved from a pattern memory, generated in real-time by a signal generation source such as a pseudo-random pattern-generator or similar logic source. Some signals, such as the framing bits, have a direct correlation to the waveform as received at the DUT, while others like the FPGA pins that pass through application

modules may not. Any signals processed such must be accounted for either as part of the process of creating the values stored in memory or within the FPGA logic itself.

5.1.3 Voltage levels

The signals as generated by this system are designed to be utilized directly coupled to the device under test, as opposed to capacitively coupling, to retain a specific DC bias and allows the use of burst signaling which is required for the Data Vortex. Because of this, the signals are sensitive to interfacing requirements with respect to signal swing amplitude and bias offset. The silicon germanium output buffers utilized on the high-speed serializer modules can be adjusted to vary peak voltage, low voltage, and midpoint bias levels over a range of levels and, if necessary, additional modifications to the signal format can be made through a redesign of the module or the use of additional intermediate processing modules.

Provisions are included on the test development platform for the lower speed framing signals and on most of the plug-in modules to allow for optional addition of capacitive coupling if the DC bias is not desired.

5.2 Data Vortex Specific Formatting

One of the defining characteristics of the Data Vortex is that it is an optical packet switched network. Compared to a statically switched network which maintains a continuous signal path from source to destination for the duration of a signaling sequence, the Data Vortex launches encapsulated packets down dynamically switched network segments. Over a dozen signals, half of which are not directly generated by the FPGA but are instead derived through the serializers, must be aligned and structured into a precise packet to be usable at the system level. The network relies on dense wavelength

division multiplexing to include an array of control, routing, and payload channels within a single optical packet.

The packet slot time utilized for the test bench is 25.6 ns and is fixed to a specific hardware implementation of the Data Vortex. This timing is dependent on the cumulative propagation time of signals through the fiber lengths connecting nodes, routing calculations, and SOA switching times [60]. At 2.5 Gbps signaling rate, this packet slot time corresponds to 64 bit periods. Each packet is indicated by the presence of a Frame signal which is present for the duration of the packet. Routing channels, the quantity of which depends on the total number of nodes in the system, carry the addressing information which is used within the Data Vortex to route the message to the desired port. These routing signals utilize the same timing as that of the Frame.

The data is 32 bits in length per payload channel at 2.5 Gbps with the clock on a separate channel extending before and after the data. Since the clock is not continuous between packets, these extra transitions are required to setup the deserializer utilized on the receiver modules and flush the final data word to the FPGA. The data and clock signals are roughly centered between the 7/8ths of the packet where the frame is valid (Figure 5.8). This leaves space on either edge of the payload channels for specified guard times intended to protect the signals from being inadvertently cut off due to non-ideal behavior within the network. This could result from such sources including, but not limited to, inexact fiber lengths between nodes and switching decision delays. However, due to the programmable nature of the signals comprising the packet structure and the payload transparency of the Data Vortex [61], the whole pattern can be shifted an integer number of bit periods in either direction independent of the frame and routing signals . Finer resolution adjustments can be achieved by utilizing the delay skew capability of the transmitter modules [70].

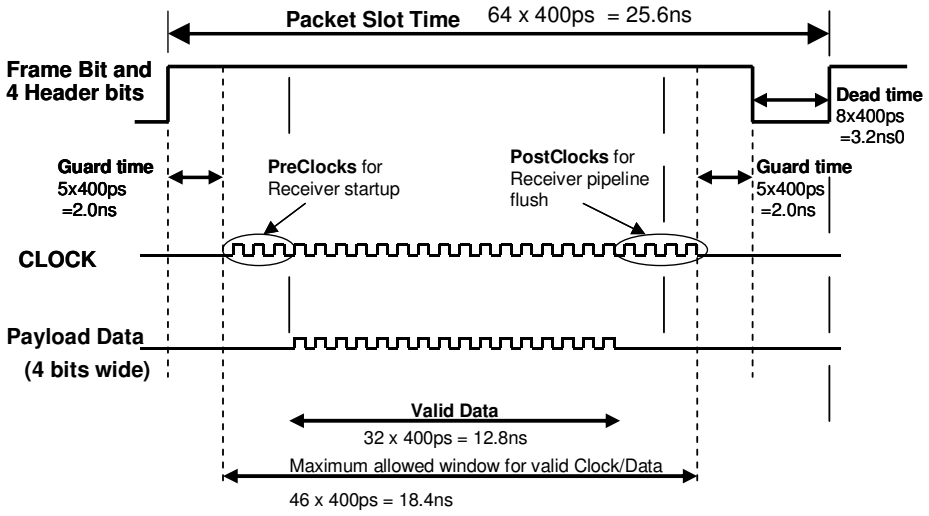


Figure 5.8 Packet structure developed for testing of the data vortex.

To assist in the capture of the packet at the destination a clock signal is transmitted in parallel with the data signals as an additional part of the packet payload. This is necessary because the very short duration of the signals combined with the packetized nature of the transmissions prohibits the use of other recovery techniques such as CDR using embedded clock information. The specific implementation of the existing 2.5 Gbps modules incorporates a deserializer chip that requires a few leading and trailing clock transitions relative to the data to setup and flush the device. These receiver modules are discussed in more detail in the next chapter. A guard band of 2ns remains to either side of the clock signal. If any changes are required to the packet structure to accommodate different transmitter or receiver modules, no changes are required to the Data Vortex as long as the fundamental timing requirements in Figure 3.4 are maintained.

Because the optical network system is still under development, no standard interface exists to connect processing resources to the network, format packets for transport across the network, or otherwise exercise the Data Vortex I/O. The original guiding requirements of the test development platform were to provide such an interface in a form capable of experimentally adjusting a variety of format and performance

metrics. These controls are necessary to evaluate the various options for control and signaling that might be used within such a system. To this end, the modular test development platform and a set of compatible application modules presented in this work were developed to emulate a prototype interface to the Data Vortex. The combined test system is designed to emulate the link between a host computer and the optical I/O ports of the Data Vortex which in turn provides low-latency interconnections between the various computers. Only a single computing system has been shown in the system diagrams shown so far, but the end application expects to support many such links.

In its first full-function implementation [71], the Development Platform used TX and RX modules designed to operate at 2.5 Gbps to support the default DV packet protocol. Slower signals, such as the Frame and Routing bits, are generated directly through the DLC, timing circuitry, and buffers implemented directly on the test platform. Faster transmitted signals such as the payload components, including eight data channels and a single clock generated and transmitted in parallel with the data, is generated through the use of high performance TX modules slotted in the A banks of the test platform. At the destination receiver, the payload signals and frame are recovered from the packet with the clock and frame being distributed on the test platform to all of the slotted RX modules in the bank B slots. Transmitting a synchronous clock in conjunction with the packet allows for near instantaneous capture of the data without the use of optical resynchronization methods [72] or lost bits while waiting for a phase-locked loop to acquire and lock to the incoming signal.

An example packet generated by the current electronics is shown Figure 5.9. The signals follow the format and timing as presented in Figure 5.8 earlier. The data is 32 bits in length, shown in the figure as an alternating pattern of 1010 over the entire length, while the clock extends before and after the data. These extra clock transitions are necessary in the complimentary receivers to setup the deserializer devices and flush the

final data word to the FPGA. The packet slot time is 64 bit periods or 25.6 ns, and the data/clock signals are roughly centered between the 7/8^{ths} of the packet where the frame is valid. This leaves room on either edge of the data channels for guard times relative to the framing signal required within the network to prevent signaling errors. While the figure shows a theoretically ideal timing relationship, it has been experimentally observed that moving payload signals earlier into the packet, eroding the guard time at the left of the image, works better through the network than centering the data in the middle of the packet slot time.

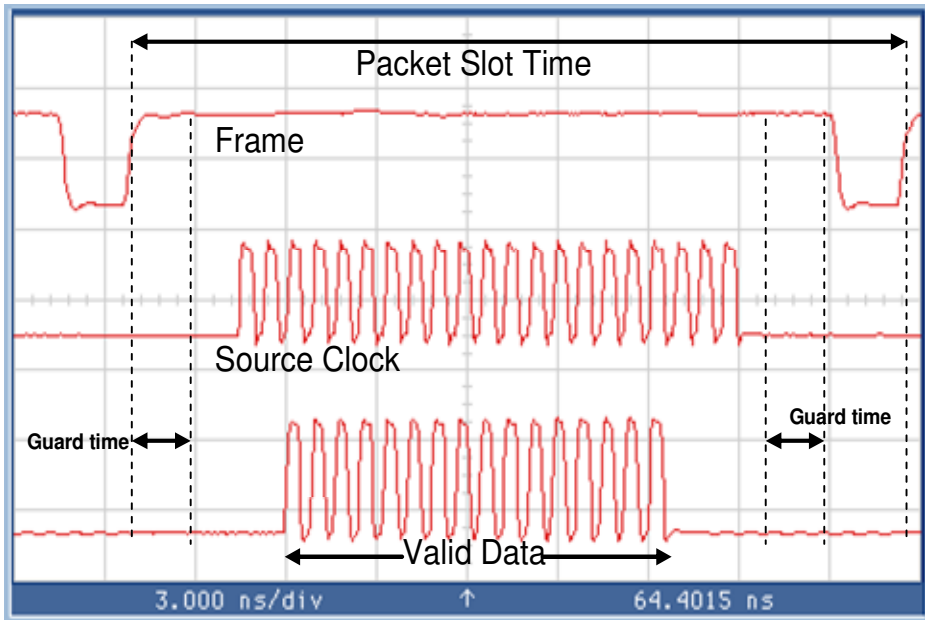


Figure 5.9 Test stimuli signals used for the Optical Test Bed application.

5.2.1 Packet Sequences

As a core feature of the test signal generation capability, signal sequences can be assembled as required, stored in memory or generated programmatically. With regards to the Data Vortex, these signals can be logically clustered into packets. A single packet or sequence of packets can be generated on demand or constantly repeated with the number of cycles between repeats controllable. Packet injection can occur constantly with the

sequencing repeating back to back or an arbitrarily long gap can be introduced to create a dead time which may affect system performance. These parameters can be varied to characterize the system performance over different loads or to create potential collision situations within the network.

5.2.2 Pseudorandom Data

Pseudorandom data generated dynamically is very valuable for testing applications because it allows for the application of nearly comprehensive combinations of input values to a DUT without requiring large quantities of pattern storage data. These pseudorandom data sequences can be created by an LFSR. Linear feedback shift registers are very small circuits which enable the generation of large quantities of pseudorandom data with very little design overhead. The circuit is implemented as a combination of D flip-flops and linear logic elements in the form of XOR gates to generate the patterns. A diagram of a small generator is shown in Figure 5.10 [32].

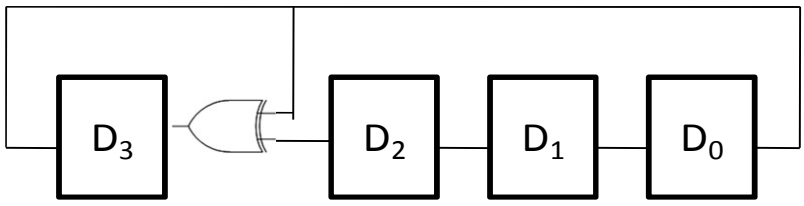


Figure 5.10 4-bit LFSR for pseudorandom data generation.

All of the eye diagrams presented in this research are generated using LFSR sourced data. The internal data path of the FPGA can be configured in a number of ways to facilitate the gathering of this data. Often a data selector mux will be placed in the path, allowing for dynamic switching between stored data patterns and the pseudorandom sequences. This mux itself can also be implemented in two different ways, facilitating either continuous run data which is useful for general signal generation applications and especially with the Data Vortex optoelectronic elements are not utilized. This mode of operation is not compatible with the packetized signaling for the network. A second

muxing scheme was implemented that allows the stored pattern data to be utilized as a positive mask. Where ones are stored as part of the pattern, pseudorandom data will be included as part of the signal burst. Output is suppressed otherwise. In this fashion signal bursts of any arbitrary shape or length can be created, up-to the maximum pattern length storable in the FPGA pattern memory.

5.3 Quantitative Results

Some representative waveforms of the system output operating at 2.5 Gbps are shown in Figure 5.11. Four data words are controlled by the DLC and serialized by the PECL circuitry. The upper most signal is alternating bits, "1010", the middle two are arbitrary repeating patterns, "11010010" and "00101011" respectively, while the last signal is "1100". The 20 to 80 percent rise and fall times are in the range of 70 to 75ps which is accomplished through the use of silicon germanium (SiGe) buffers in the final output stage.

Figure 5.11 Example 2.5 Gbps transmitter data signals for the Optical Test Bed application.

Eye diagrams were generated utilizing pseudo-random source data generated within the FPGA to demonstrate the system performance at 2.5 Gbps (Figure 5.12) and 4.0Gbps (Figure 5.13). This first data rate is the target rate for the Data Vortex test application while the latter is the maximum possible performance given this particular method of serialization and the specific PECL chips utilized in the design. The SiGe buffers introduce very little jitter which was 46.7ps peak-to-peak as measured at the crossover point on the 2.5 Gbps waveform. This results in a useable eye opening of 0.88 unit intervals (UI). The jitter measured on the 4.0Gbps waveform was comparable at 47.2ps peak-to-peak and a slightly smaller useable eye opening of 0.81 UI.

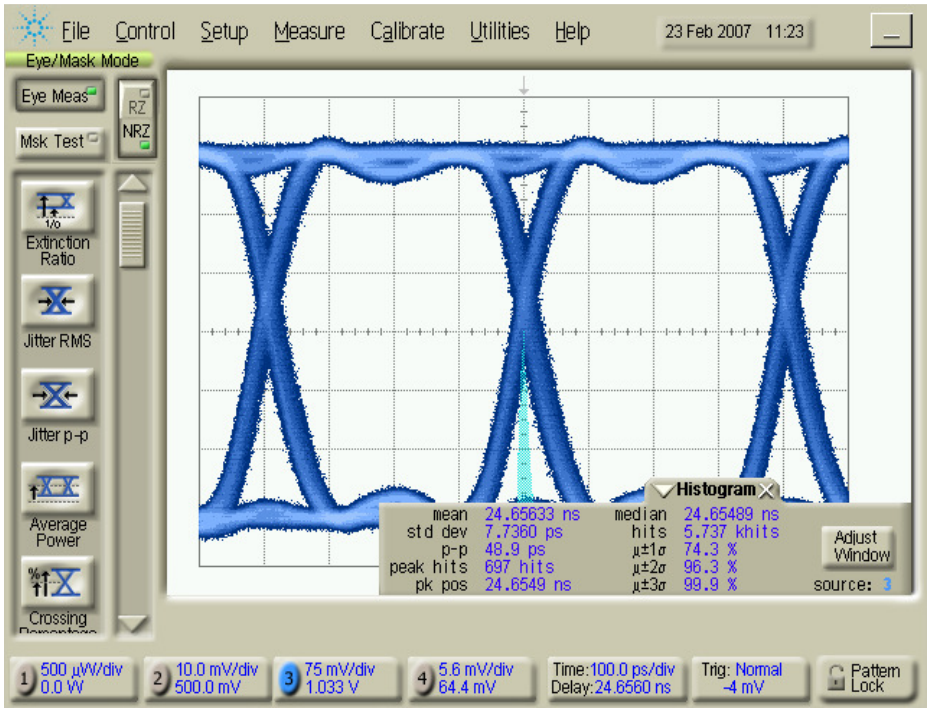


Figure 5.12 Transmitter eye diagram at 2.5 Gbps.

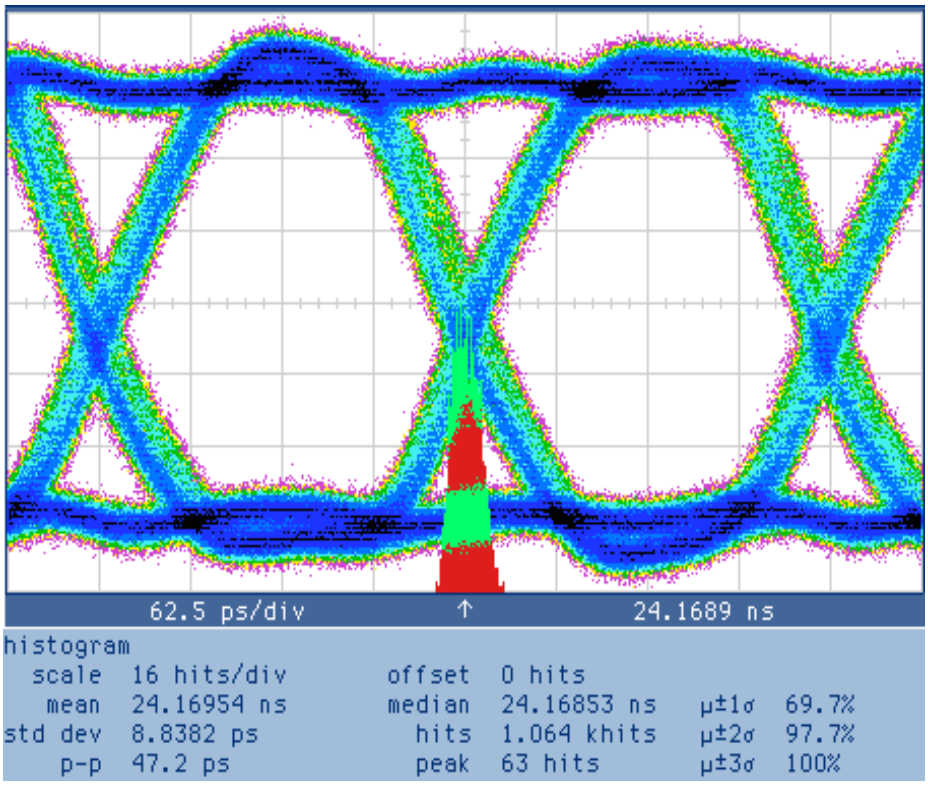


Figure 5.13 Transmitter eye diagram at 4.0Gbps.

Single edge transitions were measured on the second design platform as exhibiting 24ps of peak-to-peak jitter and 3.2ps rms jitter [1]. Unlike the previous eye diagrams, this jitter measurement does not include data dependent contributions and is therefore only related to random jitter in the reference clock used to generate the serial data stream and the contributions from any devices following the serializer component. Combined with data dependent jitter demonstrated in the previously presented eye diagrams, this performance metric limits the usable eye opening width and directly impacts the recoverability of the signal at the destination. As the total jitter increases, the data eye becomes narrower the eye and the likelihood of bit errors increases.

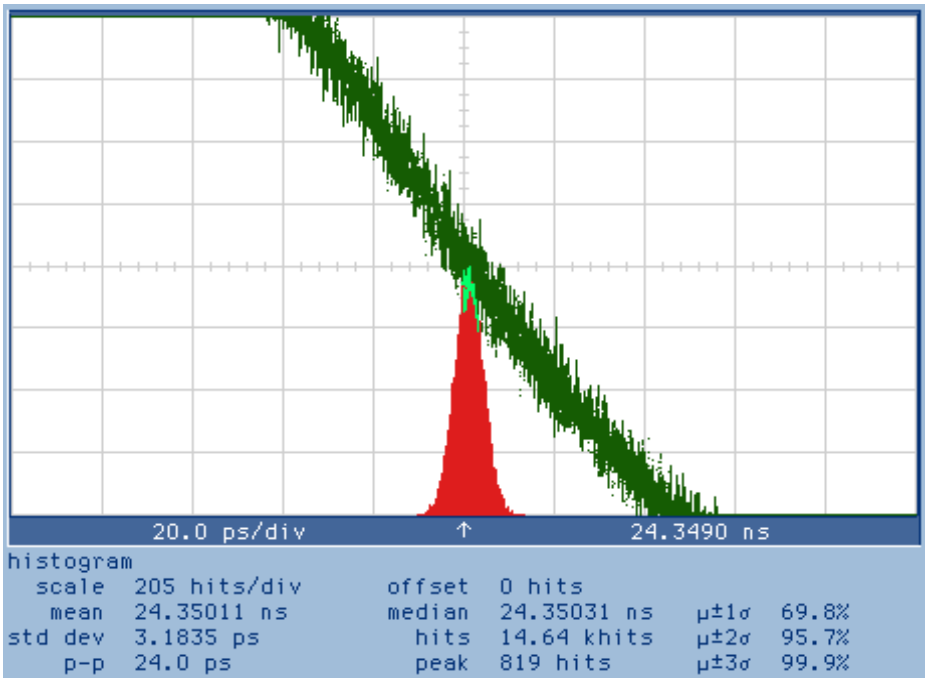


Figure 5.14 Jitter measurement for a single falling edge (24ps peak to peak) – directly integrated serialization logic on older test platform.

Incorporating design improvements from the first two designs, the current implementation of the test platform and modules has improved the signal quality shown in Figure 5.15. The improved signal edge exhibits 20.56 ps of peak-to-peak jitter and 2.77 ps rms jitter. This is accomplished through improved module layout, power distribution, and reference clock distribution on the development platform while utilizing identical chips and similar printed circuit board physical stackups.

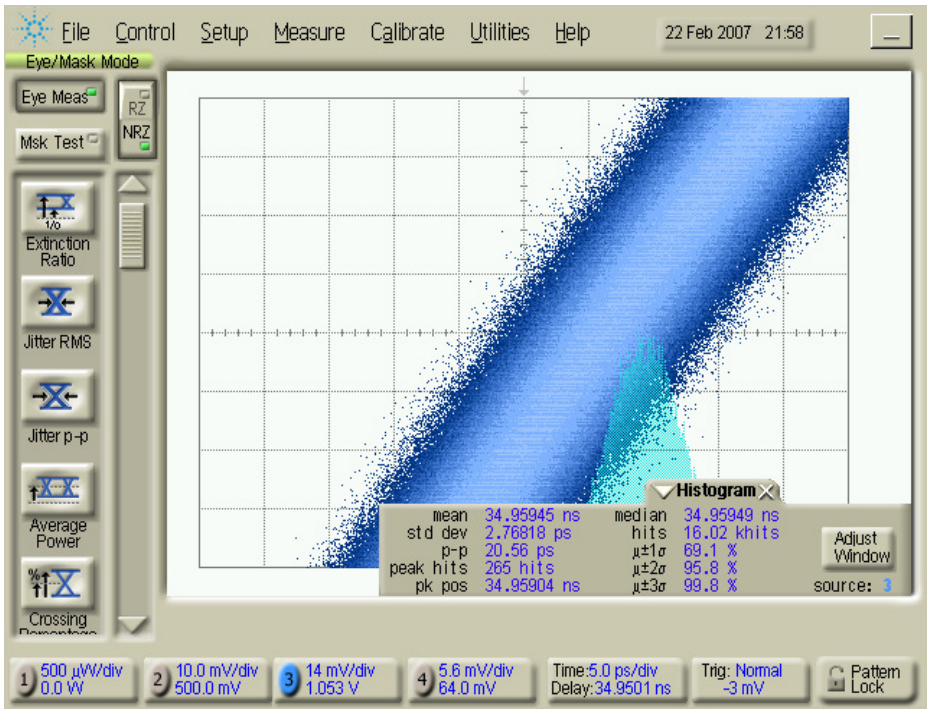


Figure 5.15 Jitter measurement on a single rising edge (21ps peak to peak) – modularized serialization logic on newest test platform.

Figure 5.16 shows a single signal edge time shifted through a range of incremental programming steps. Based on the delay calibration data gathered, ideally the signals should be evenly spaced by about 9 ps each.

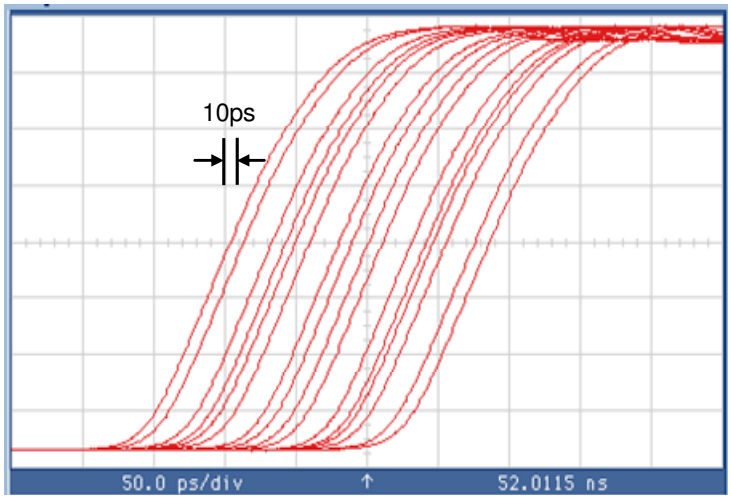


Figure 5.16 Time shifted signal in 10ps (programmed) steps.

Figure 5.17 and Figure 5.18 demonstrate the voltage variations capable using the SiGe output buffers. The first figure holds the ground reference constant and varies only

the voltage out control pin. In this fashion, the peak amplitude of the signal can be varied while holding the low reference constant, having a secondary effect of adjusting the overall signal DC offset. If the DC offset is required to remain constant, the ground reference of the buffer can also be adjusted. As the amplitude is decreased, the ground reference can be brought up resulting in a consistent midpoint.

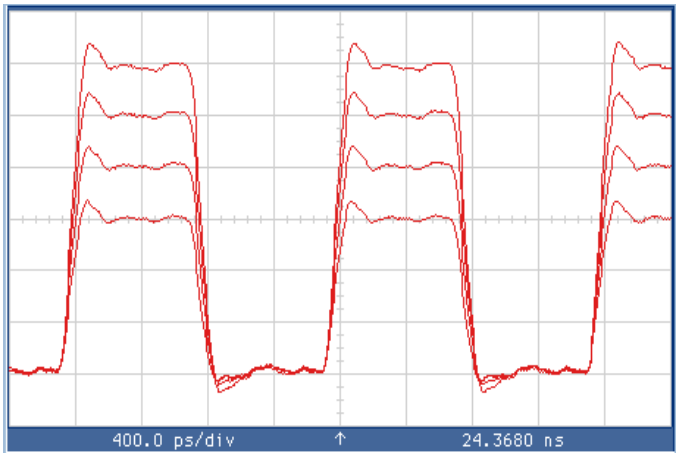


Figure 5.17 Adjusting the high logic level in 100mV steps. This example signal is running at 1.25 Gbps.

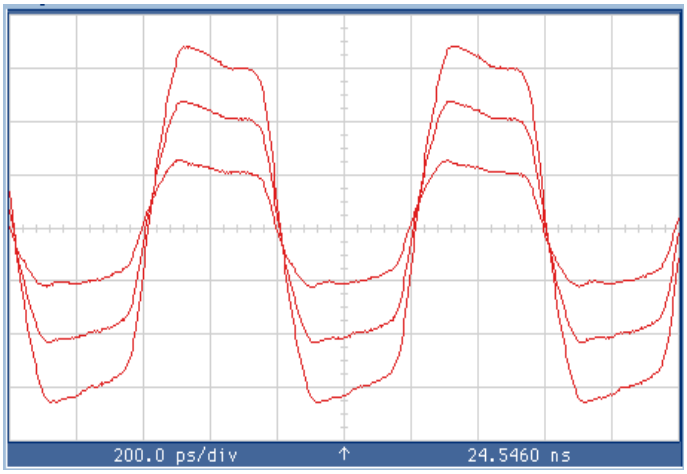


Figure 5.18 Adjusting the logic amplitude swing in 200mV steps. This example signal is running at 2.5 Gbps.

CHAPTER 6

TEST RESPONSE CAPTURE

While Chapter 5 focused on the generation of stimuli signals for a device under test, this chapter is focused on the reception and capture of the system response. An overview of the basic design approach for the creation of a receiver module capable of converting a high-speed serial stream into a format compatible with the FPGA is presented as well as demonstrating how such a module can be incorporated to capture Data Vortex specific signals.

6.1 General Capabilities

The logic for a basic receiver module is shown below as Figure 6.1. In principle, this figure is very similar to the transmitter diagram presented in Figure 5.2. A variable amplitude line driver, same as the ones utilized in the transmitter logic, is used to buffer the incoming signal from any exterior systems and ensure sharp signal transitions. A delay chip, to accommodate any system variations or to permute the signal over a range of values as part of a test, and a deserializer to convert the serial data stream to a lower speed parallel bus compatible with the FPGA timing capabilities complete the system. The deserializers are driven from a clock distributed to all the receiver modules from a common clock source. As discussed for the Data Vortex, this clock must be included at the source as part of the originally generated packet and extracted at the destination. However, any clock reference with an appropriate timing relationship or which can be made to have an appropriate timing relationship by virtue of the input clock timing adjustment capability, can be used to drive the deserialization logic. Delay chips help account for the timing delay associated with distributing the clock to all of the appropriate destinations.

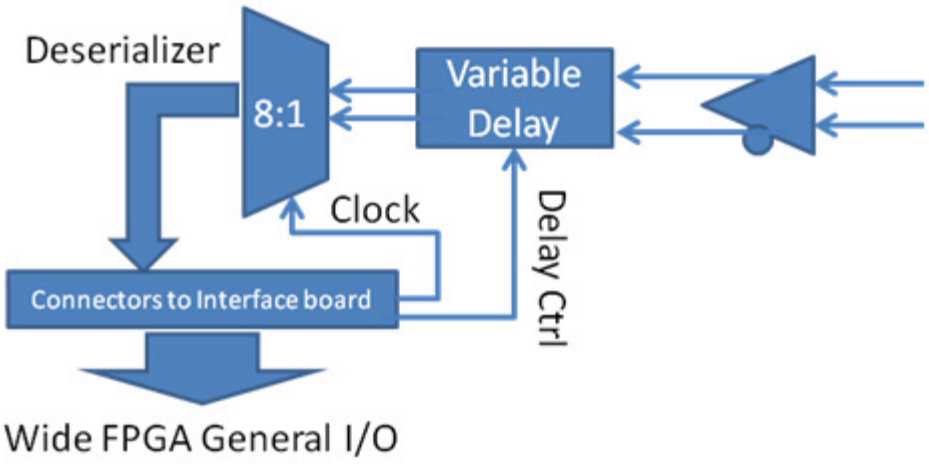


Figure 6.1 2.5 Gbps receiver module logic diagram

6.2 Test Bench Specific Configuration

A block diagram of the Data Vortex receive circuitry is shown in Figure 6.2. The payload data signals pass through 2.5 Gbps capable receiver modules as presented in the previous section. The data is passed through the same high performance buffer/drivers that are used in the output stage. The signals are then immediately passed through another delay buffer which allows for deskewing the channels individually. This helps to ensure timing accuracy or allows the simulation of non-ideal conditions within the data vortex. The packet frame signal is also utilized by the receiver circuitry, enabling the deserializer logic and resetting the system from any leftover status information that may exist in the system from previous packets. The deserializer is clocked by the delayed source clock and in turn sends a parallel clock to the DLC to sample the parallel word after every 8 bits of incoming data.

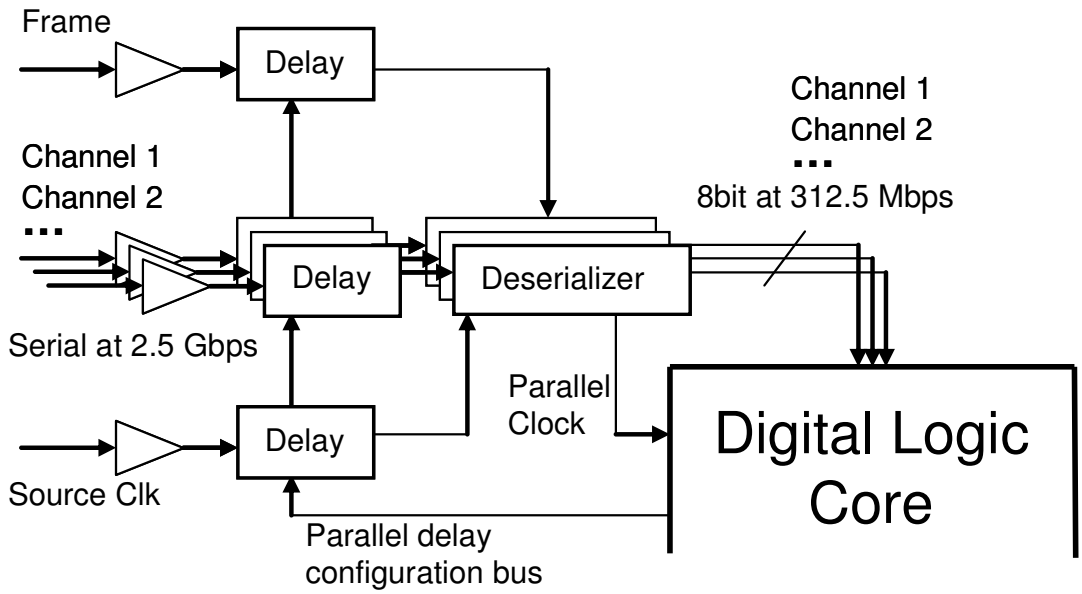


Figure 6.2 Receive Logic Diagram.

The signal timing relation at the deserializer is shown in Figure 6.3 [57]. A few clock edges (not shown) are required after asserting the frame bit to initialize the device. The chip is configured to operate on both the rising and falling edges of the input clock, recovering the serial bits and displaying the output as a parallel 8-bit word six clock periods (12 bit periods) after the first bit of data. Subsequent 8-bit parallel words appear every fourth clock period until the complete payload has been recovered by the FPGA. Due to the piplined nature of this device, an extra two clock periods are required after the final data bit (see Figure 3.3) to 'flush' the system, propagating the fourth and final data word onto the outputs and the corresponding parallel clock to the FPGA.

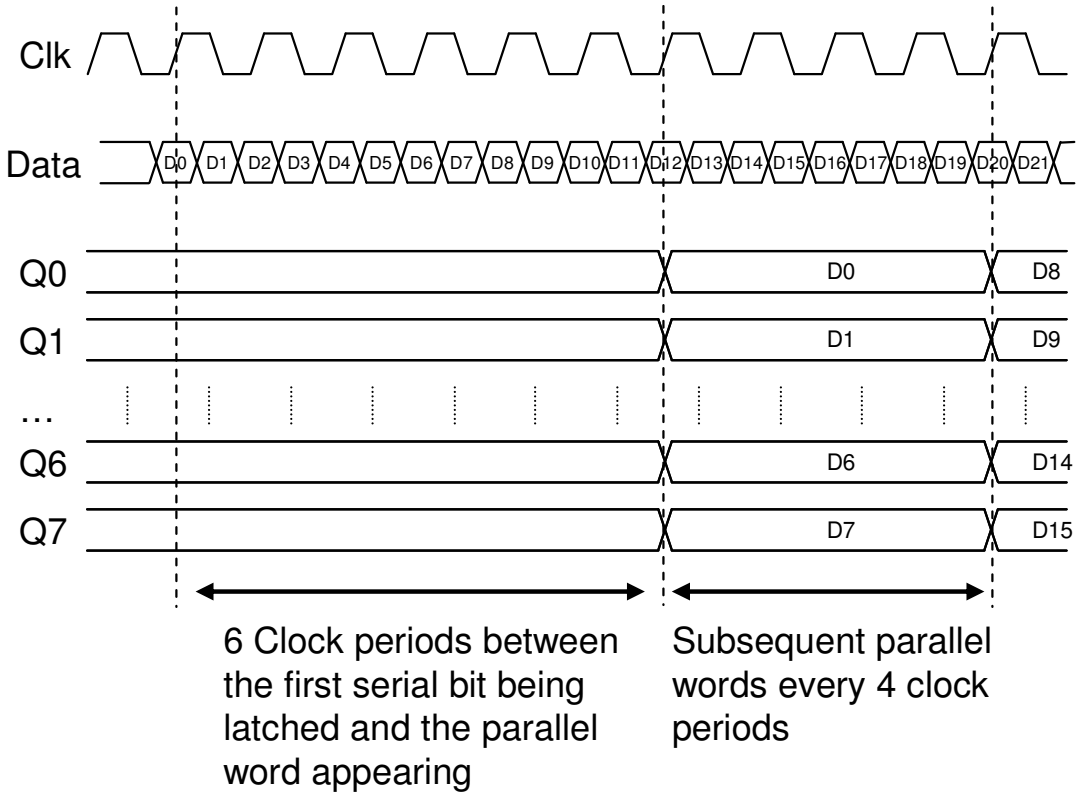


Figure 6.3 Signal timing of the deserialization process.

Figure 6.4 shows a measured signal being demultiplexed from the 2.5 Gbps serial stream to 8 bits of parallel data. The upper signal is programmed as 10101010111111110101010111111111 operating at 2.5 Gbps. The upper and lower row of arrows are indicating the first and second bit of each 8-bit word, which are shown as the middle and lower signals respectively. These 312.5 Mbps signals are read in parallel by the DLC and stored in the local memory or used in real-time testing. (Note: The parallel data has been time-shifted in the below figure. Figure 6.3 shows a more representative timing relation between the signals.)

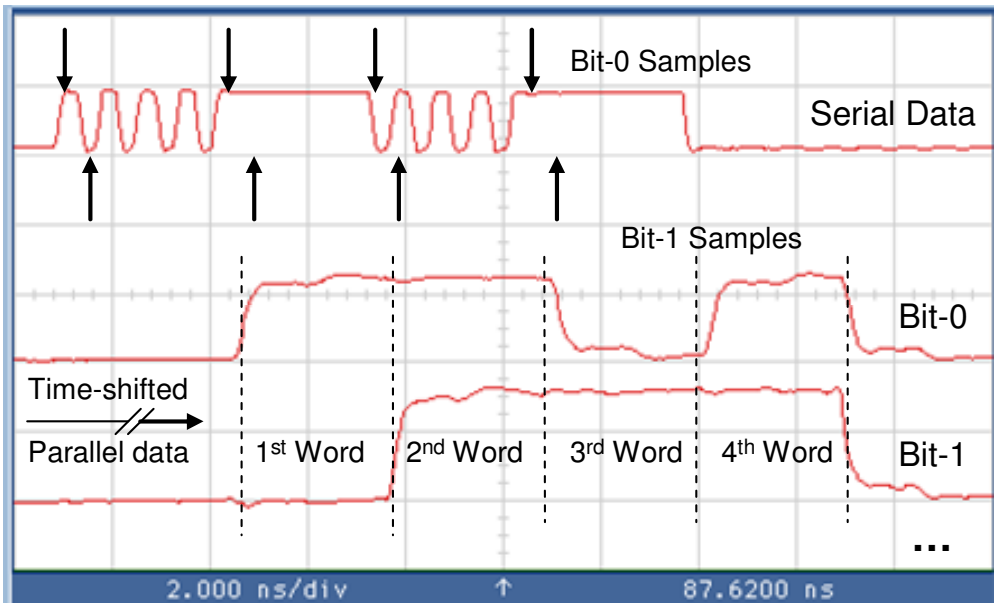


Figure 6.4 Demultiplexed serial data stream.

6.3 Results

Another timing issue that affects the optical signals is dispersion. Since the individual signals within the packet are on separate wavelengths, and the speed of propagation depends on the wavelength, the signals will drift slightly over time. This amount of drift is proportional to the distance traveled through the network. While there are provisions for adjustable time delays on all of the channels, these are designed for testing purposes and to compensate for small variations in part delays or fiber length mismatches. They are inappropriate for and incapable of real-time adjustment to compensate for dispersion effects of incoming packets. Any such adjustment would require knowledge of the number of hops taken through the system, which is not available due to the lack of a centralized control system, nor are the parts capable of adjusting at the rates required. It is therefore necessary to characterize the effect of this dispersion on the performance of the system and attempt to compensate and adjust the system to improve performance elsewhere.

Though the system uses a clock generated in parallel with the data, this clock is distributed to eight destinations at the receiver, introducing an additional source of timing variance in mismatched line lengths or part variation in the distribution tree. Combined with the reduced sampling window due to E/O and O/E conversions and dispersion as measured above, a diffused clock across eight or more channels would greatly reduce the usability and scalability of the system. Dispersion, since it is linearly proportional to the distance of travel, affects the number of hops that a packet can take which limits the total possible size of the network. Dispersion effects also grow larger as the total separation of wavelengths grows larger which would occur as additional payloads are added. Though these effects can be mitigated through design decisions and calibrated for, to an extent, even perfectly aligned signals at the input will vary in time at the destination. With a common clock across all of the channels, the final sampling window in which all the signals can be recovered is going to be smaller than that of an individual channel (Figure 6.5).

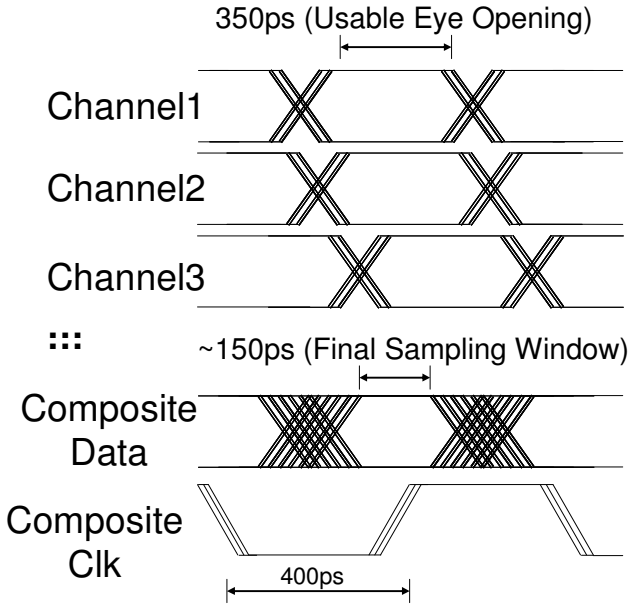


Figure 6.5 Cumulative effect of jitter and clock distribution on the data sampling window

Additional variance in the clock distribution across channels will only increase the disparity. To illustrate these issues, Figure 6.6 shows an overlay of the sampling

clock received at four of the payload channels (two signals are overlaid on each other at either end of the shown range). These signals serve as an outer bounds, with the other four from the grouping being in the same location or towards the middle of the gap. The total clock variance across the eight data channels was measured as approximately 80 ps, but the difference in two clusters of four (channels one through four and five through eight) showed the clock to be within 20 ps, suggesting possible part variation in the two halves of the distribution tree. Fortunately, since the data on each of the channels can be time delayed independently, this clock-to-data disparity can be mitigated through calibration and achieve ±10 ps alignment to an ideal loopback reference. While the skew can be tuned to satisfy the clock-to-data requirements for any one channel which minimizes this comparatively large gap between channels, this further increases the difficulty of satisfying the setup and hold requirements for all channels across the system.

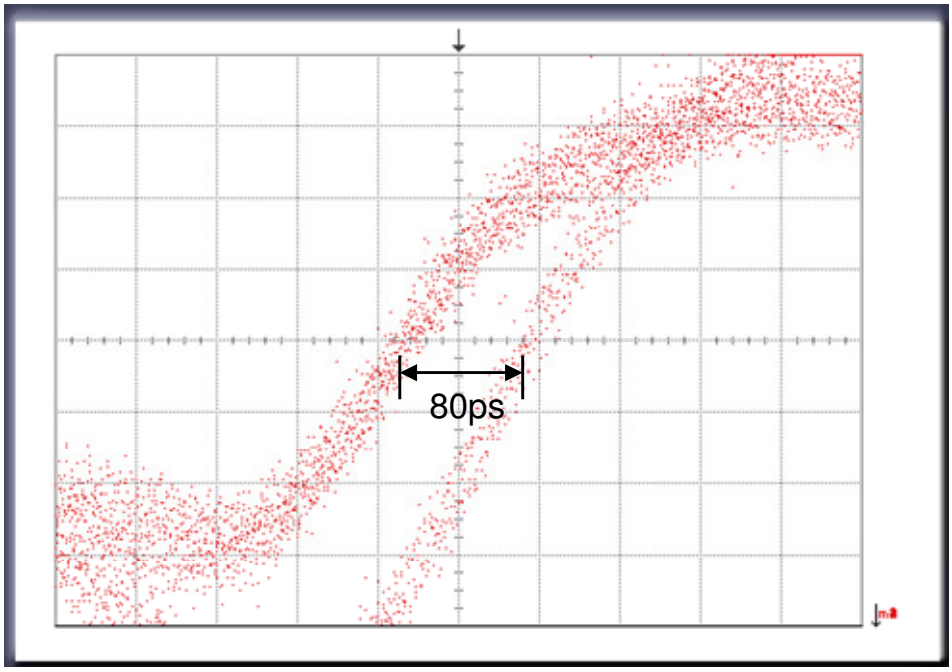


Figure 6.6 RX sampling clock, composite across channels

The clock-to-data relationship on the transmitter side however is a fixed value, depending on the layout and parts used. The reference clock distribution across the nine high-speed channels was measured as approximately 20 ps and the data from the FPGA

varied up to 170 ps among a random sampling of the 72 signals, though these 8-bit busses operate at only 312.5 Mbps and can tolerate such values.

Of the 350 ps usable eye opening, the available sampling window in which the deserializing circuitry will properly function was experimentally measured as 270 ps. This reduction from the ideal usable eye width is due to the setup and hold times required by the device. The valid sampling window was measured by using the tunable delay, skewing the incoming serial stream relative to the sampling clock. This time delay was set to the minimum value that would allow the signal to be properly decoded across its full length of 32 bits and was gradually increased until the decoded signal became invalid and the difference measured. These results were obtained in the ideal case, operating the system in a loopback configuration with a cable going directly between the SMA ports of the board. Passed through the opto-electrical components and routed through five hops in the network, the signal was reduced to a 150 ps usable timing window. It is believed that this reduction is a cumulative effect as the signal passes through the distributed printed circuit board, E/O, and O/E components as well as additional dispersion across the WDM bit-parallel message.

The amount of dispersion that a group undergoes is related to the particular set of wavelengths and the original optical spectrum was not selected with these effects in mind so the arrangement was redesigned. The physical network is tuned to a specific set of routing and frame wavelengths, so only the payload and clock signals were changed. The previous and updated spectrums are shown in Figure 6.7. The revised spectrum has been designed to reduce the effect of chromatic dispersion by centering the clock between the payload channels and condensing the overall range down to a tighter region. The nine wavelengths (eight payload + one clock) are distributed from 1543.73 nm to 1550.12 nm with 0.8 nm spacing between each channel, effectively reducing the timing skew between the outermost two channels. Additionally, by selecting the clock wavelength midway between the shortest and longest wavelength, the maximum clock-to-data timing skew is

reduced to 58 ps/km. A very large data vortex network, having 10k x 10k input and output nodes, would incorporate approximately 200m of interconnected fiber. This results in a maximum clock-to-data timing skew of less than +/- 12 ps (for SMF28 fiber with dispersion = 18 ps-nm/km) [59].

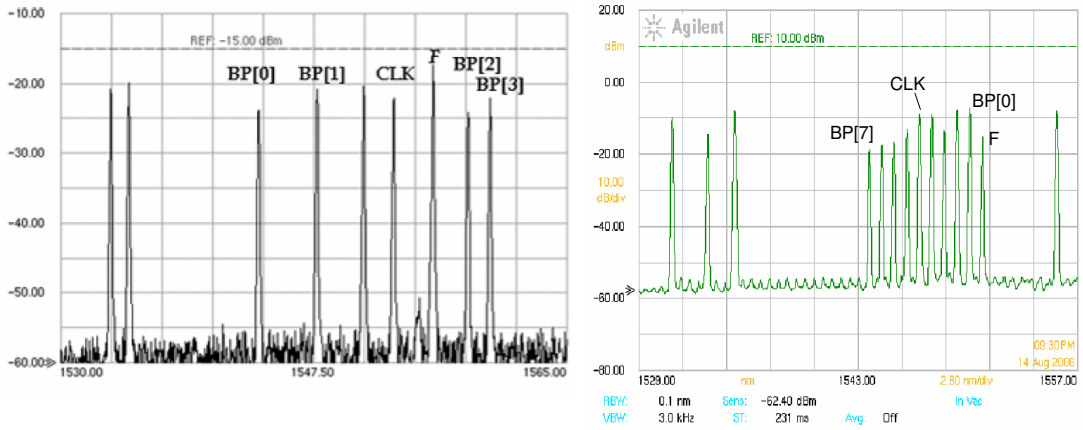


Figure 6.7 Optical packet spectrum from the previous design (left) and current (right).

This loss of timing precision was a central focus of many stages of research as it directly impacted the ability of the system to be pushed to higher signaling rates and to increase the width of the parallel payload. It was ultimately determined that a different encoding scheme, eliminating the parallel clock and creating the need for a burst capable clock and data recovery circuit, would be required for the development of any future transmitter and receiver modules.

CHAPTER 7

PROCESSING AND ANALYSIS

The test platform allows for a variety of results processing and analysis. The memory structure allows for remote processing at a low or high level of abstraction, direct real-time processing, or characterization sweeps to evaluate the system over a range of operational parameters.

7.1 Remote Processing

Using the multi-port memory structures within the FPGA, a set of memory paths can be setup to allow for data to be offloaded to an external system, typically the same one used for control and configuration, to be processed remotely. Data can be evaluated at a low or high level of abstraction depending on the information desired from the system under test. Depending on the FPGA design configuration, this change between high and low level analysis can even be performed without any changes or reprogramming of the hardware.

For instance, the smallest unit of data currently transferrable by the USB interface is a single byte. The control software can read and write to any of the universally addressable memory spaces allowing for the injection and observation of this data on a very granular scale. This allows for very precise debug and system manipulation capabilities.

At a higher level, collections of this byte data can be evaluated in bulk to evaluate the system at a much more coarse scale. In the evaluation of the Data Vortex network, this translates into the processing of one or more packets at a time to check for validity of the received packets versus those transmitted as a literal packet error rate tester. More complex injection patterns across multiple injection ports can also result in packet

deflections which ultimately alters the arrival order of packets traveling through the system which can be observed externally.

Test routines can be created, applied, and evaluated by anyone with sufficient understanding of object oriented application programming and using a behavioral level model of the target and test system. Data can be statistically analyzed directly or stored for additional processing in other software of choice.

Data throughput is bottlenecked by the uplink system interface. Even PCI Express at one lane is insufficient to keep up with the real-time throughput of a single 2.5 Gbps tester channel (unless the zero's are compacted) due to the 8b10b encoding resulting in a 2.0 Gbps data throughput and not counting overhead. It is possible to either under sample the signals to or create a small test buffer to gather a snapshot of a single test.

7.2 Direct Processing

Essentially any test achievable using the above methodology can be similarly implemented directly within the FPGA, though this requires a much more intimate knowledge of the structural design elements of the hardware. By moving the processing to within the FPGA itself versus offloading the data to an external system, the data throughput bottleneck can be eliminated and real-time processing capability enabled. This would allow for information such packet error metrics to be condensed into a form that can be transmitted across the bandwidth limited interfaces.

Some test circumstances, such as out of order packet arrival, may seriously complicate the verification logic. Ideally, test methodologies and algorithms could be developed remotely in a software environment which would allow for rapid deployment of test variants. Once the design is verified, it can be converted and migrated to a hardware implementation. Some direct processing solutions may also not be capable with the current hardware infrastructure. Once such circumstance has been identified

which includes packet verification if pseudo-random injection cycles are executed from more than one test system into the network at once. With no hardware level cross platform synchronization available, it may be impossible to verify correct reception of the packets generated by different sources or to even maintain synchronous injection between the various test platforms.

7.3 Alignment

In addition to the bit-wise alignment issues discussed in Section 2.3, there could be byte alignment issues in a run of data samples. Figure 7.1 shows a payload channel, at the top of the image, from a packet that is transmitted repeatedly, back to back using the 2.5 Gbps TX and RX modules. The received data is stored in a byte buffer that is shown at the bottom of the image. While the packet logically contains only 4 bytes, one additional byte on either side of the data burst is captured by the FPGA as a side effect of the deserializer behavior and was partially illustrated in Figure 6.2. There is a time lag between serial bits entering the device and parallel data being available for capture. As a result, assuming ideal timing between all of the respective data, clock, and frame reset signals, a single empty byte will precede the data. Similarly, excess clock transitions follow the final serial bits to ensure that the deserializer is flushed, resulting in another empty byte trailing the data. When viewed in sequence, the data from each packet appears in clusters of 6 bytes but wraps around on the various rows due to the display grid being 8 wide.

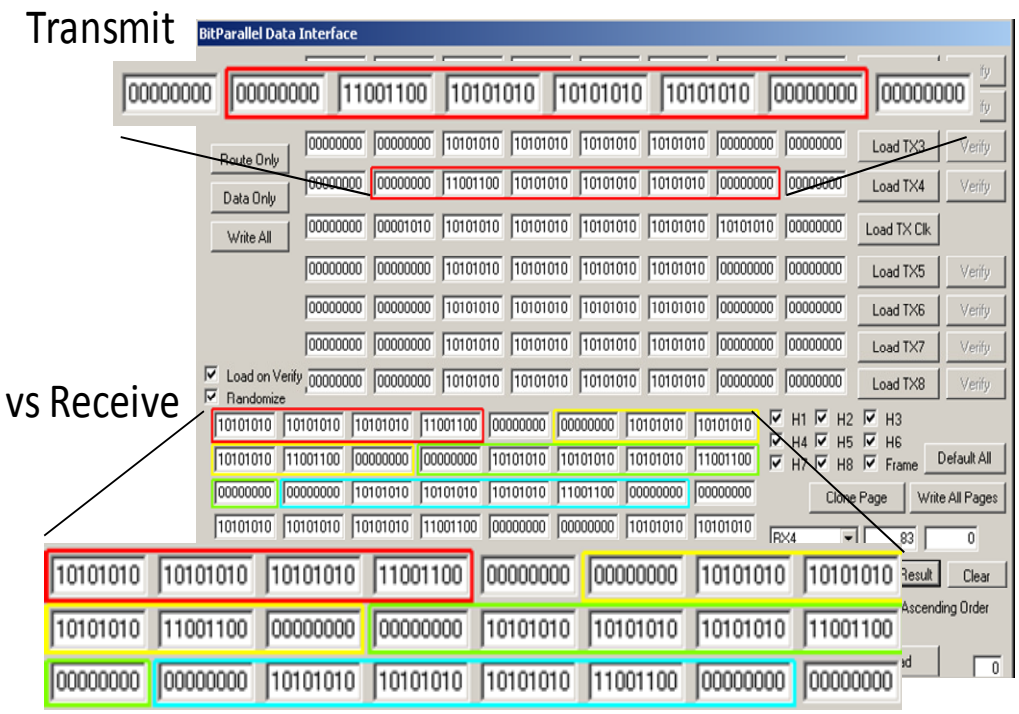


Figure 7.1 Byte-wise data alignment in received data.

Note that the transmitter box displays the sequence as it is transmitted in time, increasing left to right. The received data is actually displayed from a shift buffer, newest data first. The lead byte for each of the packets, 11001100, therefore appears at the right of each cluster of 6 bytes. Knowing this particular behavior of the receiver modules, this version of the software incorporates a set of options that allow the exact same data to be displayed in a slightly different fashion. Figure 7.2 shows this same test executed with a Stripe value, corresponding to the expected number of bytes captured from a packet, of 6. The Ascending Order checkbox is also disabled for this second image, allowing the displayed packets to be read left to right identical to the transmission order shown on the screen.

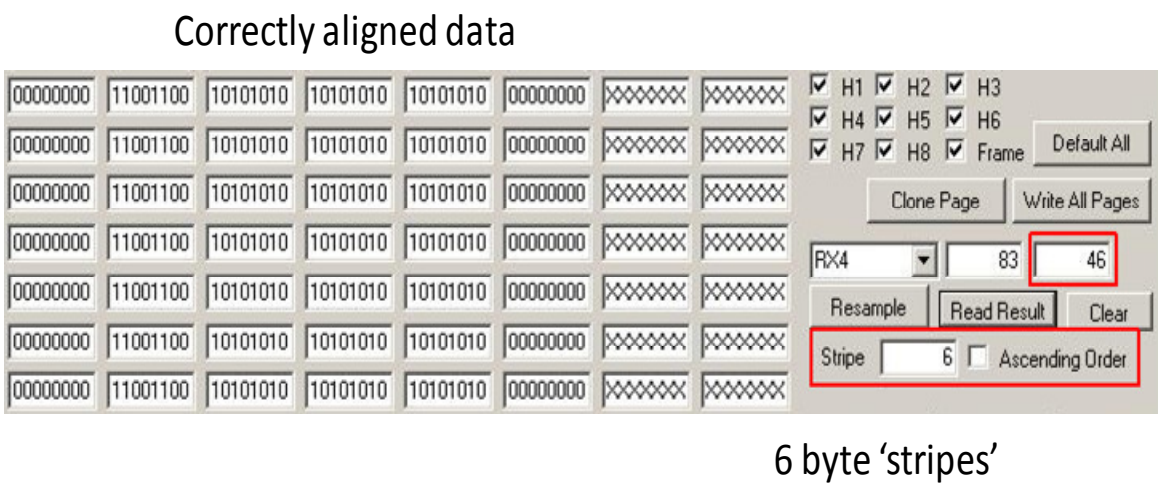


Figure 7.2 Correctly aligned 6 byte striped data.

The sampled results are identical in value across both images, but are easier to read and comprehend by a user in the second. The particular striping value can be set by observation or based on knowledge of the system parameters. The amount could even be detected automatically. By bracketing the data with zeros which can be used as a key signature, an algorithm could be devised and executed in software on the host computer or within the FPGA to adjust the alignment.

7.4 Characterization

One of the primary performance metrics which needs to be evaluated with the packet protocol used in this research is the width of the valid sampling window. This width affects the maximum signaling rate achievable with this specific implementation. The width of this sampling window can be evaluated by finding the general alignment, assisted through manual alignment or through a search algorithm in software, and then skewing the signals in time earlier and later to find the sampling margins. This process has been conducted in various incarnations of the design to detect how much of the 400 ps theoretical window is usable, in the ideal electrical loopback case and through the network as displayed earlier in Chapter 6. TX skew can also be artificially introduced to

emulate non-ideal system circumstances or to experimentally deviate from the theoretical normal values.

This chromatic dispersion or group velocity dispersion (GVD) [73,74] is the effect of signals encoded on different wavelengths of light propagate through a medium at slightly different rates. This effect shows up in the resulting signal as clock-to-data skew across the payload channels. Since the alignment of the parallel clock to the individual data pulses is critical to the sampling and recovery of the data, this skew is one of the largest controlling factors regarding the current scalability of the network. Of the 400 ps bit period, the usable eye opening at the injection port is approximately 350 ps. Of this opening, a valid sampling window of 150 ps was experimentally found when routing a packet through a nominal number of nodes. This window was defined as the amount of tolerated clock-to-data skew while still recovering the complete bit-parallel data. A simulated shift was added by varying the delay before the deserializer as necessary [62].

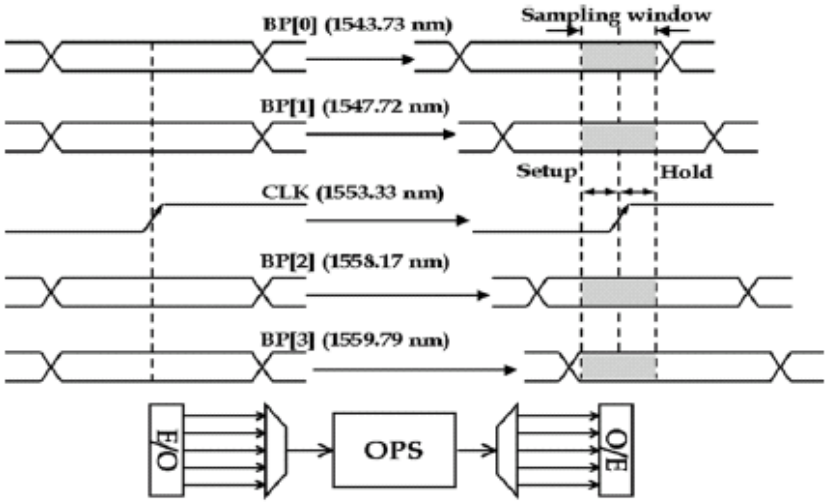


Figure 7.3 Clock-to-data skew induced to the bit-parallel messages by the OPS interconnection network.

In a system where GVD dominates the skew contribution, this sampling window suggests that the maximum scalability of this network is over 500 m of interconnected fiber. For comparison, a 10k x 10k data vortex network would incorporate approximately

200 m of interconnecting fiber. There are however other contributing factors that would need to be considered on a larger scale network, many of which are electrical. For instance, the signaling rate of 2.5 Gbps while significant when this project was begun is fairly modest by current standards. Going to a higher data rate will shrink the usable eye opening, which will result in a corresponding shrink in the sampling window. Also increasing the payload to additional wavelengths will increase the difficulty of maintaining alignment of payload channels relative to the sample clock.

7.5 Function Execution

As previously mentioned, the host computer uplink is usable for control and configuration in addition to being used as a data transport medium. Some of this behavior can be implemented by writing data, such as configuration data, to specific memory addresses and letting the internal FPGA application logic act accordingly on the information. Other behavior can actually be programmed into the FPGA as an operation or a macro 'instruction' designed to complete a complicated operation with a minimum of user intervention. These command instructions can be very simple or complex as needed and are critical to implement higher level protocols and system interactions.

Figure 7.4 shows a very simple state diagram for function execution within the test platform DLC. The default state of the system is the Standby state. A memory location, accessible by software over the USB interface, is periodically fetched and evaluated. If the value is non-zero this indicates that there is a function to be executed, otherwise the system returns to the standby state and continues the cycle. Once the function completes execution, the memory address is cleared and the system returns to the standby state. Once a non-zero value is written to this memory address, the value can be observed externally by the software to indicate when a new function execution request can be issued to the tester or indicate completion success to the user. Some examples of these functions are presented in the next two sections.

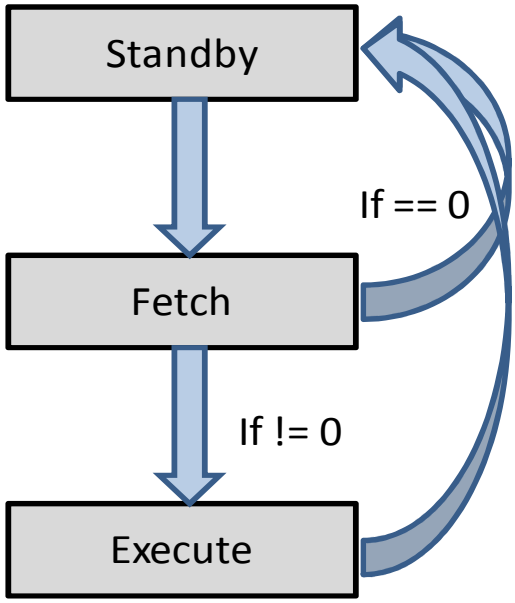


Figure 7.4 Function execution state diagram.

7.5.1 User Initiated

A simple function that may be user initiated, mentioned many times throughout this work as an integral feature for both transmitter and receiver styled modules, is timing adjustment. Figure 7.5 shows a region of the timing delay and configuration options sub-windows of the control logic for one of the network test sets. Each of the system application channels are individually represented in this interface window and have independent time skew capability, if which Header1, TxFrame, Tx7, and Tx8 are visible here. The numeric control fields indicate the amount in nanoseconds that the signals will be delayed and updates occur when the user clicks the Write Delays button. Once the button is pressed, the program begins a process which will result in the desired adjustment. For that adjustment to occur, the software must translate the desired delay values, converting them to two individual bytes which will be placed into the FPGA memory at specific memory addresses corresponding to the channels to be adjusted. Those data values will be subsequently scanned from memory, concatenated into a single

10-bit programming word, augmented with the encoded slot address, and output from the FPGA onto the respective signal pins. The delay values are presented in parallel to all of the slots simultaneously but only utilized at the slot for which the signals are intended by virtue of a slot specific enable pin that becomes active by virtue of the encoded slot address which has been decoded by circuitry on the test interface board.

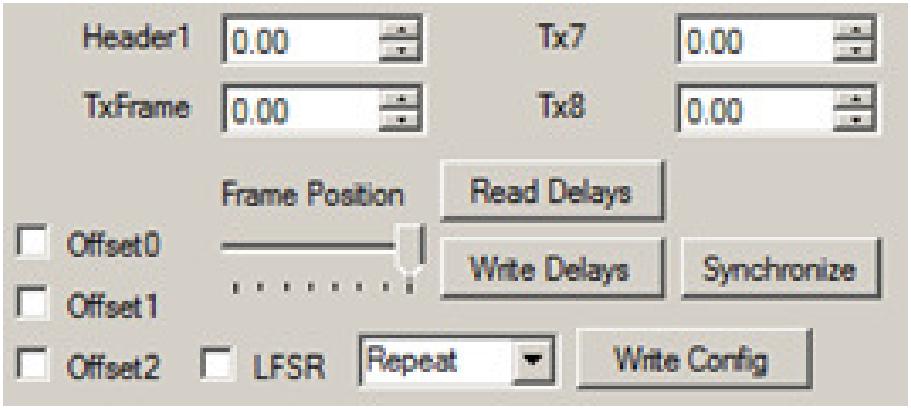


Figure 7.5 Time delay and configuration software options.

This is an intricate and complicated sequence of events but the test system user does not have to know the implementation details. As far as the software programmer is concerned, there are two fundamental steps that must occur to force a timing update in the test system: 1) Delay programming values must be transferred from the controlling software and stored within the FPGA in the appropriate memory locations and in a specific format. 2) A 'delay write' instruction is executed, which is itself a data value being programmed to a specific memory address. Any module conforming to the pinout described in Chapter 4, supporting the 10-bit bus and slot enable, will be compatible with this instruction. Specific implementation details regarding the memory locations, data formats in those memory locations, and quantity of delay chips supported will be specific to a test platform hardware implementation but will be very similar across designs. The software programmer does not have to be aware of any of these hardware specifics and simply have to place the data in the correct spots and execute the desired instruction.

There are a number of these instructions that can be implemented within the FPGA that have been developed as part of this research. One of these is also visible in the figure, specifically the Synchronize button. When clicked, the interface software downloads a different value into the reserved instruction memory address. On execution, this instruction will simultaneously pulse a differential synchronization signal to all of the Bank A slots. How this signal is utilized, or if it is utilized at all, will vary depending on the specific modules in the slots. For the 2.5 Gbps transmitter modules, the synchronization pulse resets the sequence counters in the MC100EP446 8:1 serializer chips utilized on the module, ensuring that all of the transmitter modules are aligned. Internal to the FPGA, secondary logic resets a number of other sequence counters, control bits, and any other logic that is subject to being potentially desynchronized or corrupted over time.

7.5.2 Protocol Emulation

These feature executions, while they can be detailed as a series of memory write and read operations initiated by hand by a knowledgeable user or automated through software, a more robust solution is to 'teach' the test system about a particular protocol and allow the transactions to be handled automatically. An example of this approach was demonstrated through an experiment designed to emulate a packetized memory transaction from a source DV node to a distributed memory system accessible through a different DV node. Memory transactions, which were simple read or writes, were structured into a specific packet protocol, transmitted across the network, received, and processed accordingly. The signal flow within the system boundary of the test system is shown in Figure 7.6.

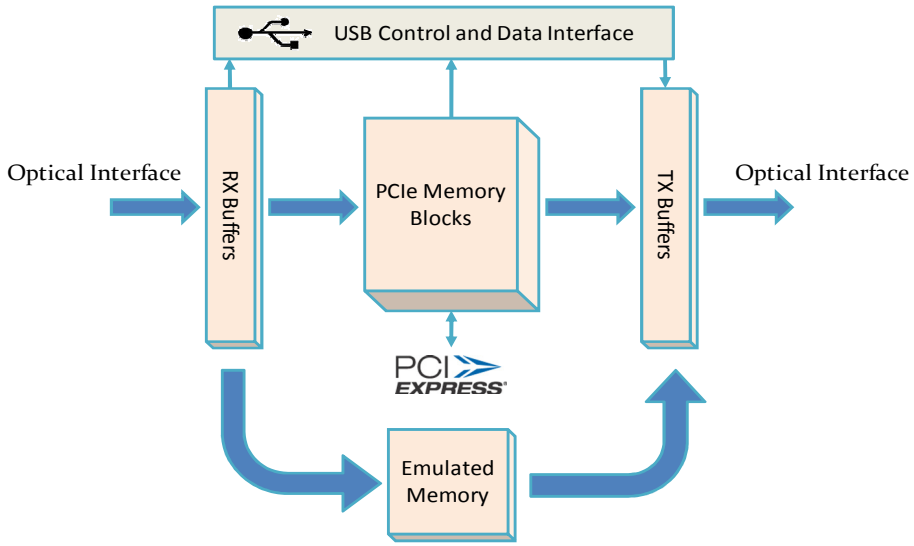


Figure 7.6 Memory transaction protocol emulation.

An individual memory transaction packet supported by this system is shown in Figure 7.7. The address to be written or read is stored in the first byte, sequentially in time, of the fourth payload channel. The second byte corresponds to the type of transaction where the least significant bits, indicated as TT in the figure, are 01 for a write or 11 for a read. The remaining bits are ignored and were included for future expansion if necessary. All of the remaining bytes in the packet, including those from additional payloads, are treated as data.

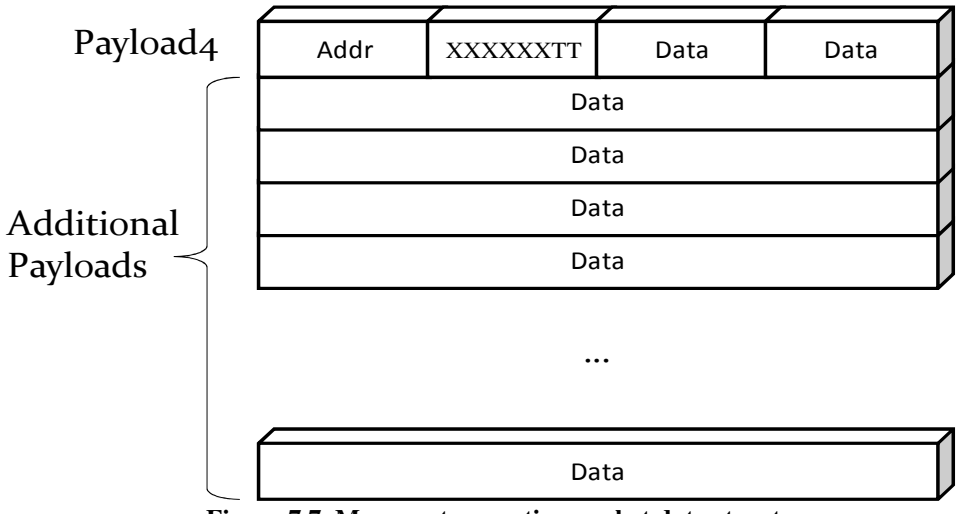


Figure 7.7 Memory transaction packet data structure.

As the system operated, new packets were retrieved from the PCIe interface, converted to this packet structure, and transmitted to the desired destination on the network. At the destination, the address and transaction type were processed and acted upon accordingly. For a write operation, the data received in the packet was written to the appropriate address and the transaction was considered complete. For a read, the data region of the packet is overwritten with the data from the desired address. The restructured packet was then forwarded to the transmitter to be resent through the network. For this particular experiment, only one test system was utilized, mimicking both of the nodes at either end of the transaction so a constant set of routing bits were utilized. In a more complete system, information such as the transaction source could be included in the packet, including those previously ignored bits in the transaction type bit, which could be used to return the transaction results to the original requesting node.

CHAPTER 8

MODULE AND APPLICATION VARIANTS

The test development platform, while originally designed with the specific 2.5 Gbps Data Vortex test requirements in mind, has incorporated a number of features and design specifications covered in Chapter 4 to allow for alternate or expanded applications. The test platform features can be exploited to support alternate module implementations for generation and capture of higher frequency signals, create variant channel configurations, or provide additional system utility. By using these new modules exclusively, in conjunction with older modules, or varying the FPGA programming the test platform can support a wide variety of new test applications or extend the current one into the future. Several new modules have been developed that can be combined in a variety of ways within the Development Platform. The design and performance, in the case of the particular modules that have been manufactured as shown below in Figure 8.1, are discussed in detail in the following sections.

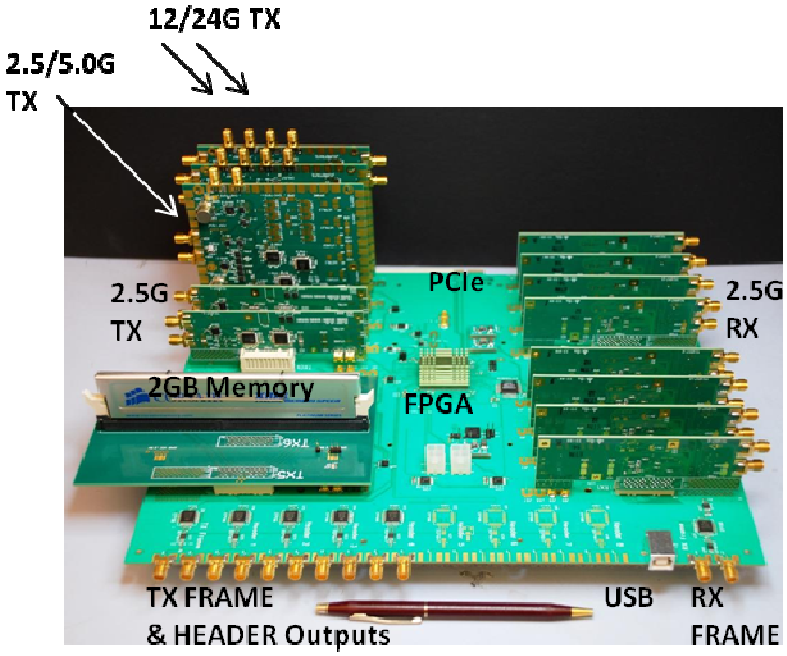


Figure 8.1 Photograph of Development Platform with 2.5/5.0G and 12/24Gbps modules.

It is worth noting that the configuration shown in the picture is for demonstration purposes only and the majority of the slots (everything on the right hand side and two on the left) are filled with earlier-generation TX/RX modules. However, the remaining slots demonstrate a few of these newer modules such as the 2.5/5.0Gbps transmitters, 12/24Gbps transmitters, and the memory utility module. Differing module categories can be mixed as needed and this flexibility is described in more detail in section 8.4.

8.1 Dual-channel 2.5 Gbps TX/RX Modules

During the design of the Development Platform, extra resources were included in anticipation of future testing needs. These can now be used to significantly extend the functional capabilities of newly-developed modules. Specifically, each of the 9 "Bank A" slots in the platform has 18 moderate-speed (up to ~800 Mbps) signal wires connecting to the central FPGA controller. Test signal data or control words can be passed to/from the modules across these parallel busses. In the original DWDM application, only 8 of these were used for parallel data to/from the 8:1/1:8 TX/RX modules. Since the existing

busses are more than twice as wide as originally needed, a natural expansion would be to double the number of serializer/de-serializers on each module. Two high-speed channels per module can also support a further 2:1 multiplexing to obtain even higher speeds (>5Gbps). Furthermore, the new modules include high-performance "RF" relays for switching external DC test signals into the DUT connections, while maintaining signal integrity for multi-GHz functional tests. Block diagrams of the new dual-channel 8:1 TX and RX modules are shown in Figure 8.2 and Figure 8.3 respectively.

The dual-channel 2.5/5.0 TX logic shown in Figure 8.2 is divided into two identical data paths at the top and bottom of the figure. The basic function of these paths are identical as the earlier single channel modules, including an 8-to-1 multiplexing logic from the FPGA, delay circuit and variable-amplitude SiGe buffer. In addition to this base logic, the new designs incorporate double pole double throw (DPDT) relays on the outputs. These relays allow for application of high-speed signals generated by the module to abe applied to the DUT as at-speed, multi-GHz tests. Alternatively, the relay may connect moderate-speed signals for lower-rate functional test or DC tests. Since the RF DPDT relays have about 15 GHz bandwidth, they pass this wide range of possible signals with very little distortion.

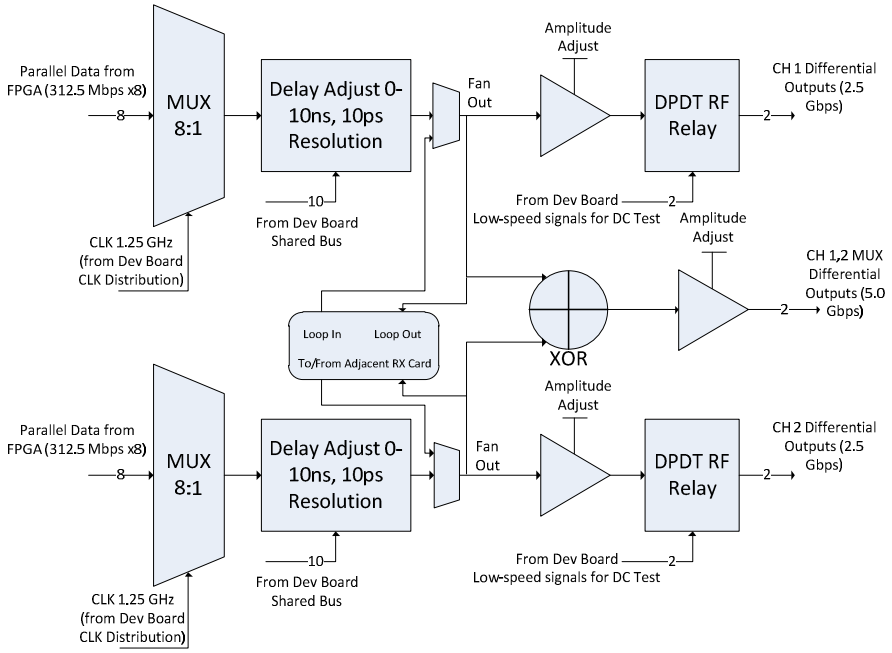


Figure 8.2 Block diagram for the dual-channel 2.5/5Gbps 8:1 TX module.

Beyond the use of the high-performance relays, each channel includes an InP exclusive-OR gate, with variable-amplitude adjustment. By adjusting the two channels with a ½ bit-period delay offset, the XOR gate can be used to obtain a double-data rate signal. Therefore each of these modules can produce a multiplexed signal running double the rate of the individual channels or up to about about 7 Gbps.

Other new features of the dual-channel TX/RX modules include "loop-back" paths for either the DUT or the ATE signals. To complete these paths, one TX module is connected to an adjacent RX module using blind-mate z-axis coaxial connectors. Of course, this requires that the Development Platform be configured with alternating TX/RX modules. The new RX designs accomplish the 1:8 demultiplexing functions, and otherwise share similar characteristics to the TX modules.

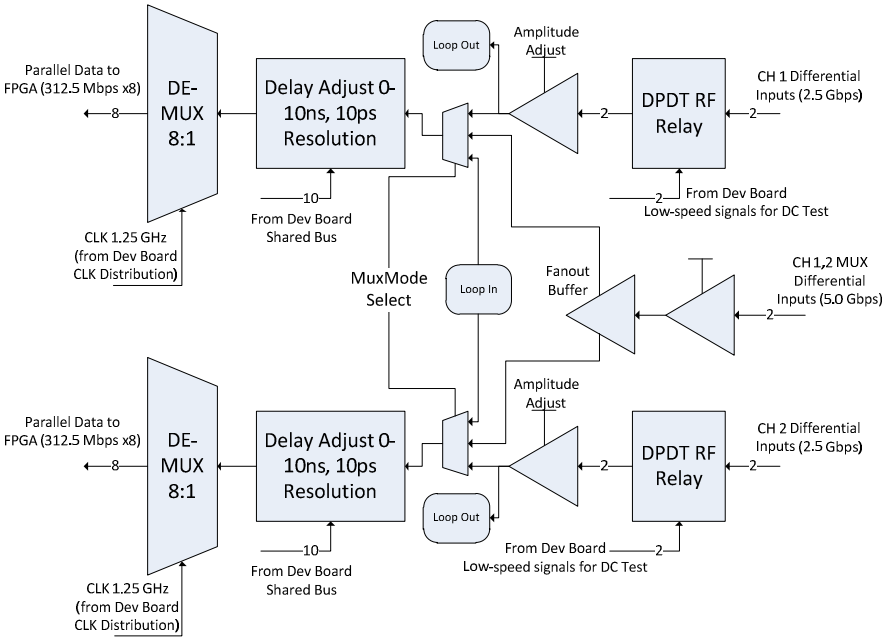


Figure 8.3 Block diagrams for the dual-channel 2.5/5Gbps 8:1 RX module.

Because these designs are directly based upon the earlier, single-channel modules, it was expected that their performance would be comparable to the earlier prototypes. In fact this proved to be the case. Example data eyes from the new 1:8 TX module are shown in Figure 8.4 and Figure 8.5, measured at the output port corresponding to the XOR channel multiplexor. An individual channel can be observed at a time by generating a string of zeros on the opposite channel or two independent streams can be generated and skewed in time to achieve the desired multiplexing. The single channel performance can be directly compared to the results from the single channel module shown in Figure 5.12. The signal quality on this dual-channel module is consistent with the previous results with the new module having slightly improved edge transition speed due to the high-performance XOR device.

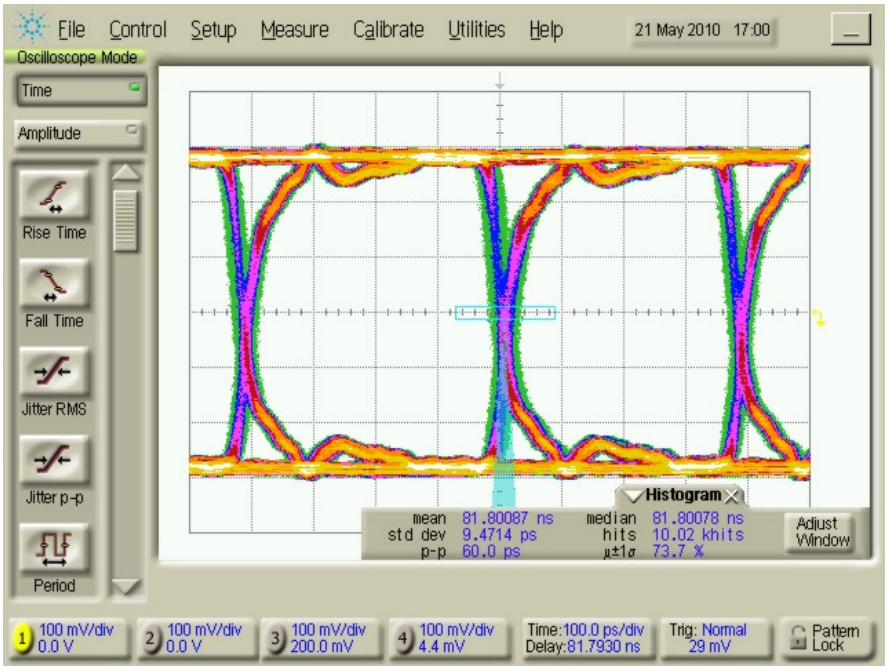


Figure 8.4 Performance of the new module at 2.5 Gbps

The multiplexed mode, shown in Figure 8.5, is capable of operating at exactly double the independent signal rate. As a result, the module can generate maximum rates up to 7 Gbps, though 5 Gps is shown to compare to the 2.5 Gbps eye shown previously. The signals show very good "open-eye" characteristics, with 20-60ps total measured jitter (for ~10kHits). More detailed measurements showed that this total jitter is made up of about 25ps data-dependant deterministic jitter (DJ), and about 3.5ps (RMS) random jitter (RJ). From this we estimate that the approximate total jitter at BER=10^{-12} is TJ = 14.1RJ + DJ = 75ps. Therefore even at this BER, there is a data-eye opening of 62.5% unit interval (UI) at 5 Gbpsm calculated by dividing the reduced bit-width by the total bit width or 125ps/200ps. At 2.5 Gbps, the data-eye is 84% UI or 325ps/400ps.

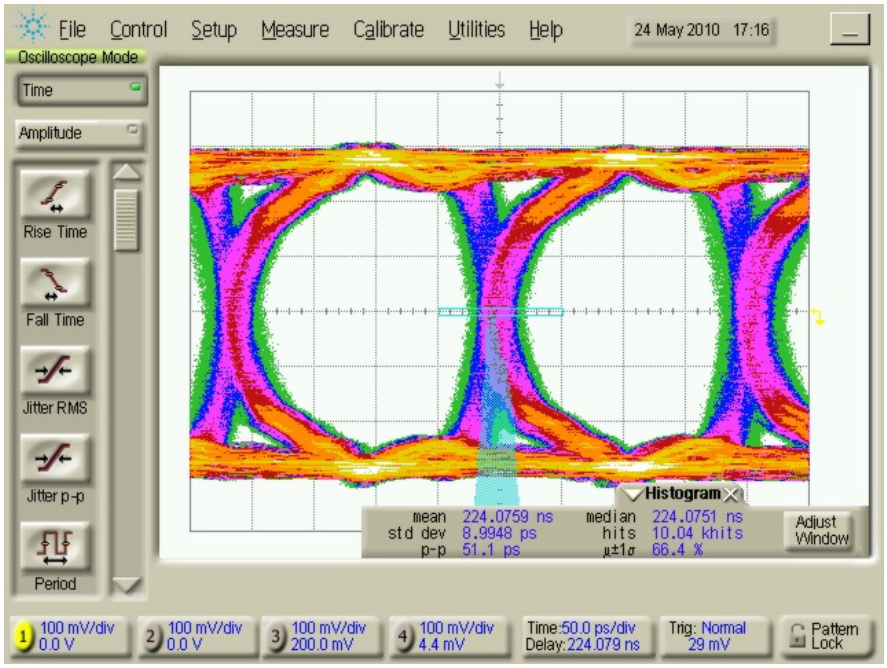


Figure 8.5 Dual channel card, multiplexed to 5.0 Gbps.

The dual channel RX modules discussed here have not actually been constructed because the current test platform is incapable of properly supporting their full functionality in the types of burst-mode signaling situations being studied in this research. The Bank B connectors are incapable of supporting the full 16-bit parallel word width. The Bank A slots can support this data width, but is not configured to support two different clock references, which would be required for independent TX and RX functionality, or the distribution of a bank wide packet/frame reset signal. Continuous or near continuous run systems which don't require a packet reset would theoretically be compatible with this module design as populated in the Bank A slots of the current system, assuming a timing reference could be achieved using the clock common to all of the banks. A test platform re-design incorporating additional parallel bit capability to the FPGA would similarly resolve all of these design constraints.

8.2 12 Gbps TX/RX Modules

In a desire to achieve signaling capability even beyond the 5 Gbps multiplexed capability of the modules in the previous section, a 12 Gbps capable system was devised. A simplified diagram of the design and signal flow for a 12 Gbps capable transmitter module very similar to the other transmitter designs presented so far is shown in Figure 8.6. Whereas the 2.5 Gbps modules directly phase shifted the serialized data stream, the stream generated by the 16 to 1 serializer is beyond the maximum capability of the delay circuit used throughout this work. However, the same timeshift capability can be achieved through adjustment of the reference clock.

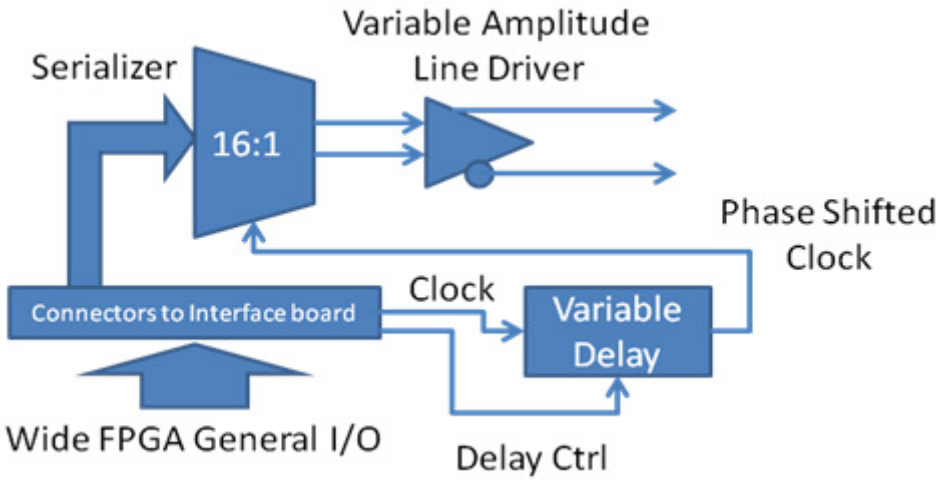


Figure 8.6 Block diagram for the single channel 12 Gbps 16:1 TX module.

The new 12 Gbps TX and RX modules (see Figure 8.7 and Figure 8.8) each utilize 16 of the available 18 wire connections to provide 16:1 multiplexing of data to/from the central FPGA. As in earlier prototypes, we use SiGe serializers/de-serializers to obtain extremely high-quality serial signals at rates up to 12 Gbps. These components are further augmented with ultra-high bandwidth variable amplitude buffers, multiplexers, fan-outs, and RF relays as shown in the respective figures. Furthermore, these modules support optional jitter-injection through the use of voltage-to-delay modulation and have high bandwidth (26 GHz) connections for "loopback" and ATE self-test/calibration.

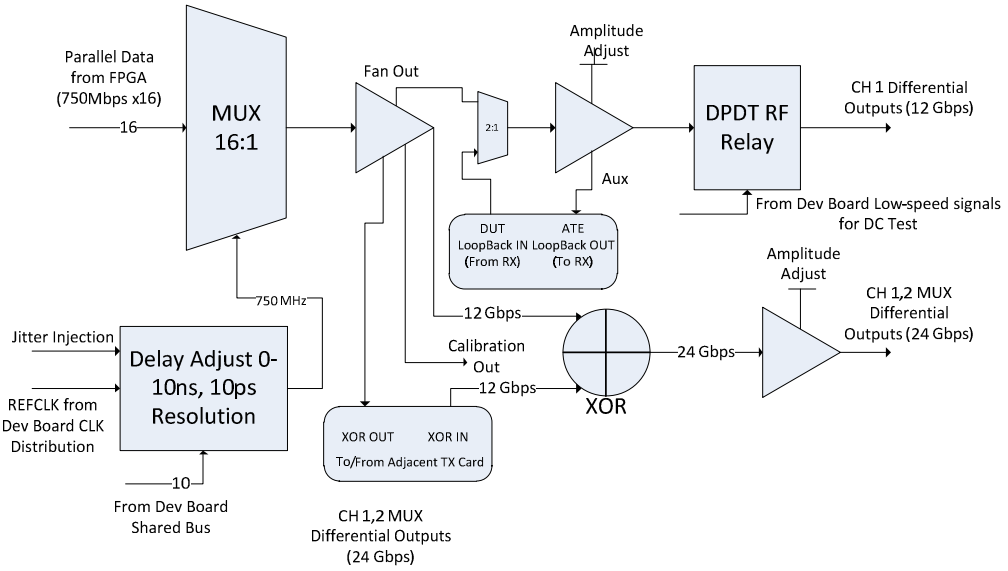


Figure 8.7 Block diagram for the 12Gbps 16:1 TX module.

The loopback paths require that TX and RX modules be located in adjacent slots in the Development Platform. Alternatively, two TX modules in adjacent slots can be used (with card-to-card connections) to perform additional 2:1 multiplexing above 20 Gbps. Similarly, two adjacent RX modules can be used to obtain 1:2 de-multiplexing of 20 Gbps signals.

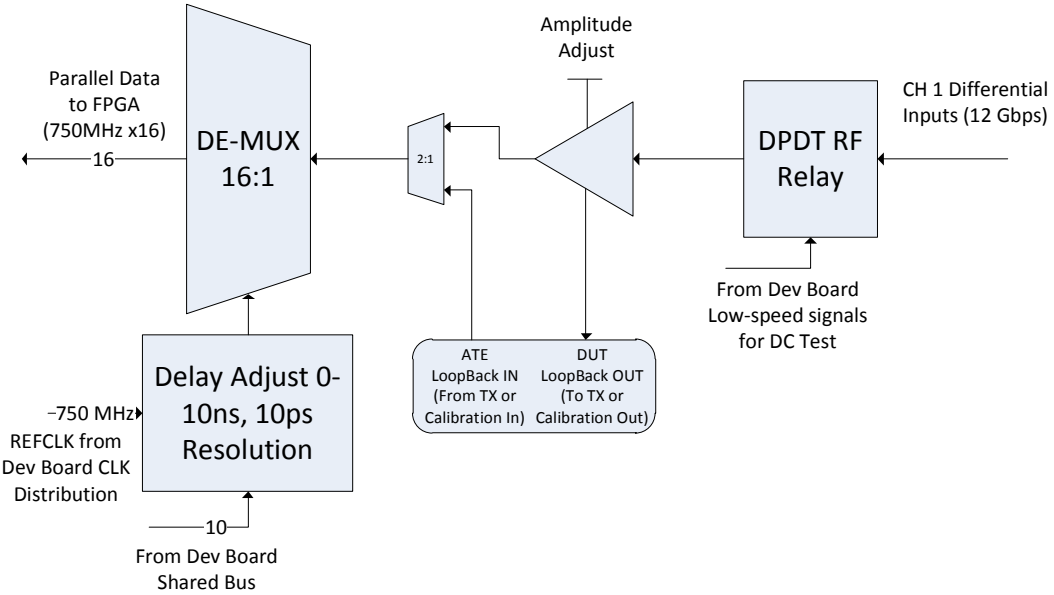


Figure 8.8 Block diagram for the 12Gbps 1:16 RX module

This receiver implementation relies on the use of the built-in CDR capability of the demultiplexing device. As a result, this particular implementation is not compatible with burst systems like the Data Vortex

A typical output from the 12 Gbps TX module primary output (RF relay output) is shown in Figure 8.9. This data "eye" pattern shows the relative timing stability as the circuit is cycled through all 2^8-1 pseudo-random bits in the test pattern used. The figure shows the performance at 12 Gbps. The measured peak-to-peak jitter (approximately the "total" jitter) is TJ = 23.6 ps and includes both random-jitter (RJ) and deterministic-jitter (DJ).

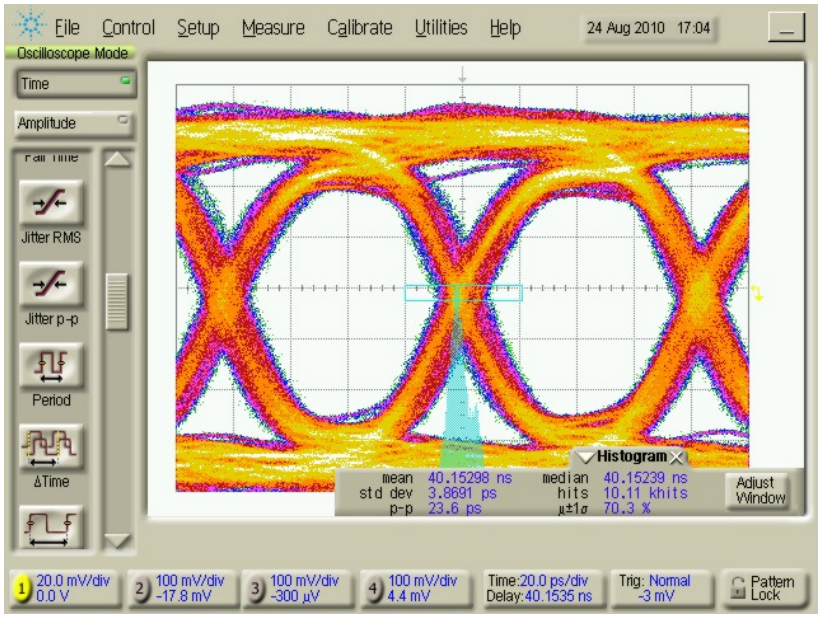


Figure 8.9 12Gbps TX module performance through output relay.

The RF relay used in the new modules is intended for 12 Gbps data applications, but is known to have limited bandwidth (approximately 14 GHz). Therefore, the signal edge-rates shown in Figure 8.9 are degraded as compared to the expected performance of the output buffer alone (expected tr=20ps, 20-80%). In parallel with the primary signal output that passes through the RF relay, an auxiliary output is provided that bypasses the relay. For comparison, that output is shown in Figure 8.10. Here slightly faster edges are

observed, but the signal shows some undesirable effects ("roll-off" following the initial step transitions). These effects are attributed to the use of relatively lossy FR-4 material in the prototypes, and also to some non-ideal matching at the SMA launch. Both limitations will be corrected in the final version of these circuits.

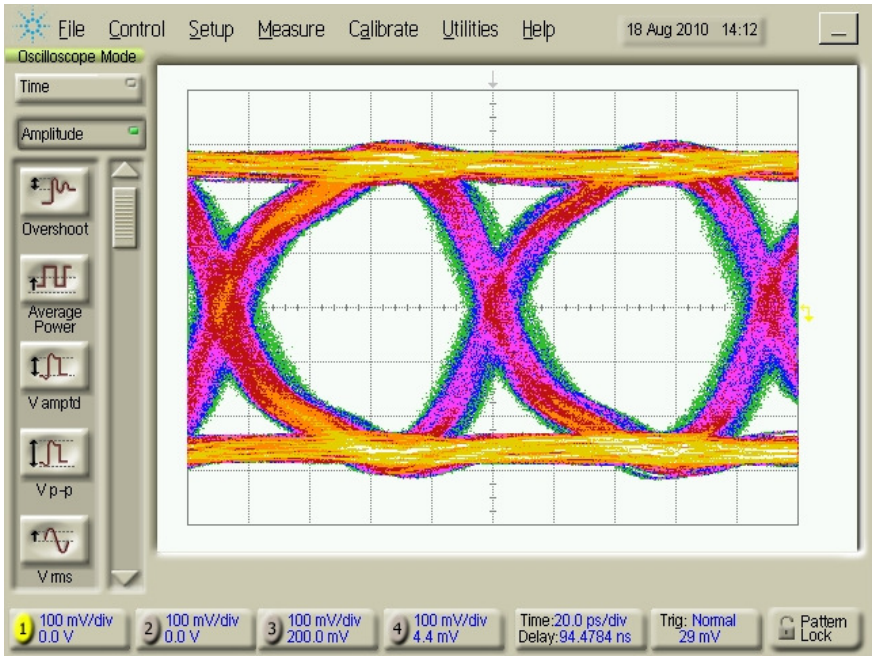


Figure 8.10 12Gbps TX module "unbuffered" performance at 12Gbps, bypassing the RF relay.

Both the Primary and Auxiliary outputs have adjustable signal amplitudes. These are adjusted using an analog reference voltage under DAC control. Examples of four different amplitudes are shown in Figure 8.11, spanning a range of about 100 mV to 500 mV. The outputs are differential, but measured and displayed single-ended. The corresponding differential amplitudes span 200 mV to 1000 mV. Notice that the data eye remains open throughout this range, with consistent timing and voltage characteristics. Total jitter is not greatly affected by adjustment of the signal amplitude. Therefore, these signals are useful for testing input sensitivity of future DUTs.

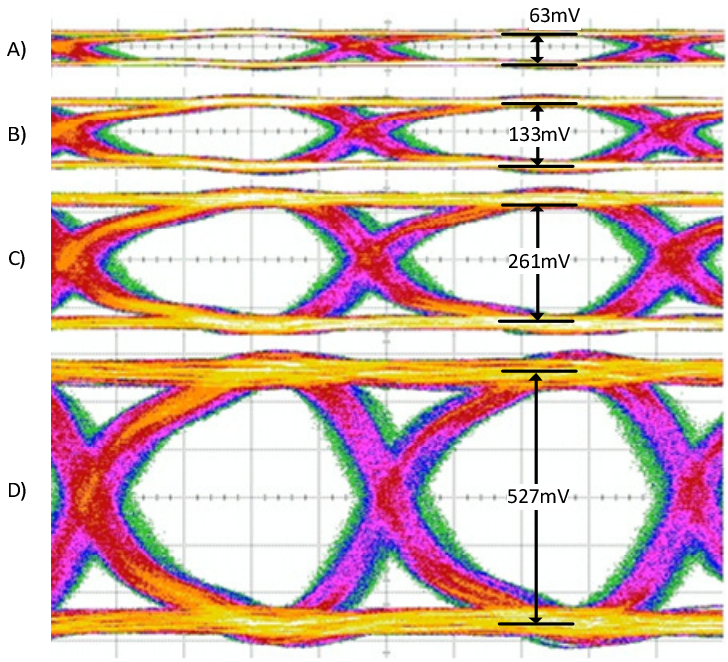


Figure 8.11 Four increasing values of Vr. Horizontal scale is 20ps per division. Vertical scale is 100mV per division.

Measurements of signal amplitude (single-ended) versus control voltage are shown in Figure 8.12. The amplitude is a nearly linear function of control voltage throughout much of the range. Similar to the timing-calibration discussed earlier, the signal amplitude must be calibrated to account for part-to-part variations. However, only a few calibration points are needed to accurately predict the signal amplitude. The DACs provide <1 mV resolution, while typically the amplitude must be controlled to an accuracy of tens of millivolts.

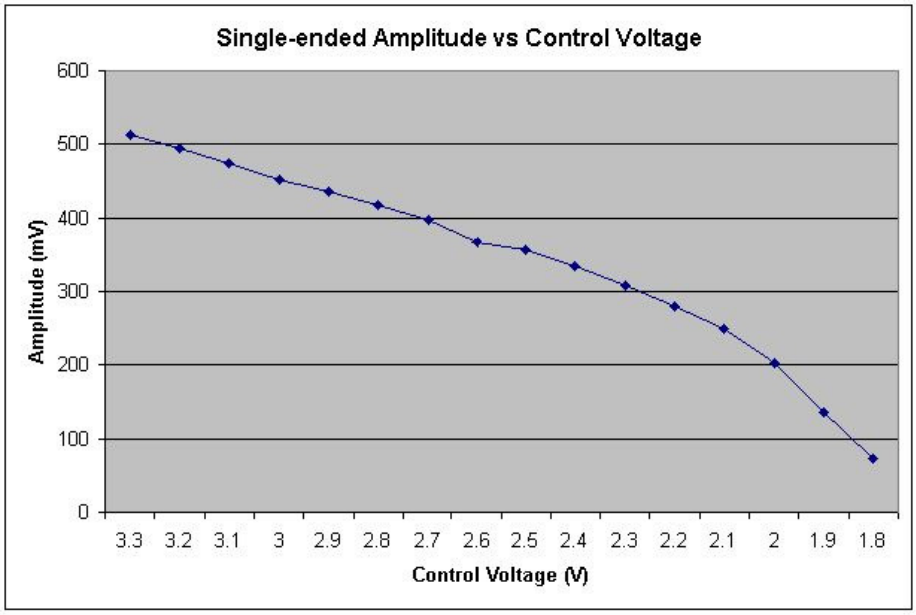


Figure 8.12 Output amplitude relative to Vr.

Because there are well-known effects that limit the Primary and Auxiliary output edge-rate and signal quality, an additional buffer was added to the output stage to "sharpen" the logic transition edges. Of course, the added buffer introduces a small amount of additional jitter (about 2 ps added to TJ). However, as illustrated in Figure 8.13, the overall signal quality is greatly improved. Rise and fall times are nearly symmetric and close to the 20 ps specified transition times (20-80%), with nearly-linear characteristics between 20% and 80%. The small increase in jitter was not noticeable in this particular measurement, but was measured using a finer time-scale.

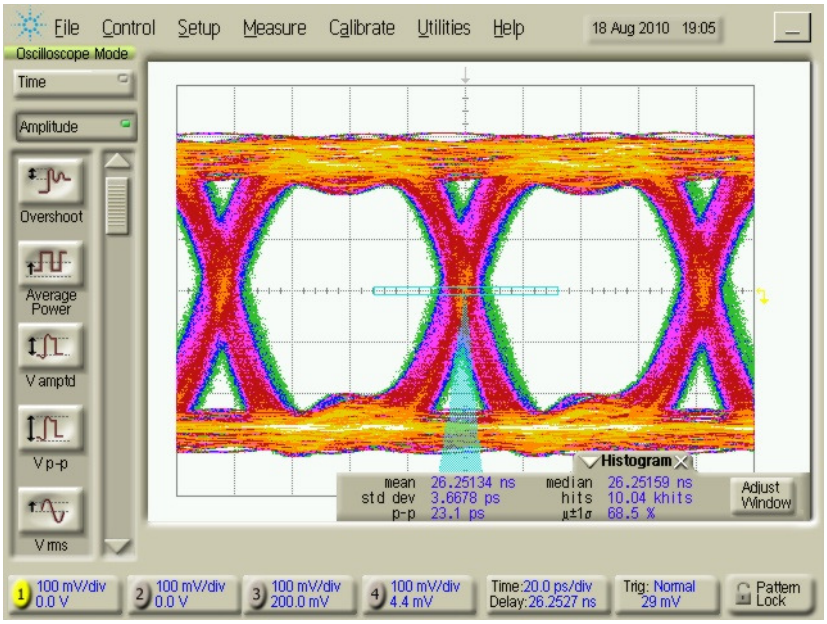


Figure 8.13 12 Gbps TX module externally buffered performance at 12 Gbps.

Amplitude adjustment is available using the external buffer (similar to the on-board Primary and AUX output adjustment). Two examples are shown in Figure 8.14 (300 mV buffered output) and Figure 8.15 (26 mV buffered output). The 300 mV amplitude is about mid-range, and representative of the "normal" range signals. However, the 26 mV amplitude is near the extreme-minimum for this buffer. In fact it is well-below the device minimum specification. There is noticeable signal-quality degradation. Nevertheless, the data-eye is still "open" and could, in principle be used for small-signal testing purposes. Keep in-mind that these are actually differential signals, so the other half of this single-ended signal provides an effective doubling of amplitude, and a larger eye opening.

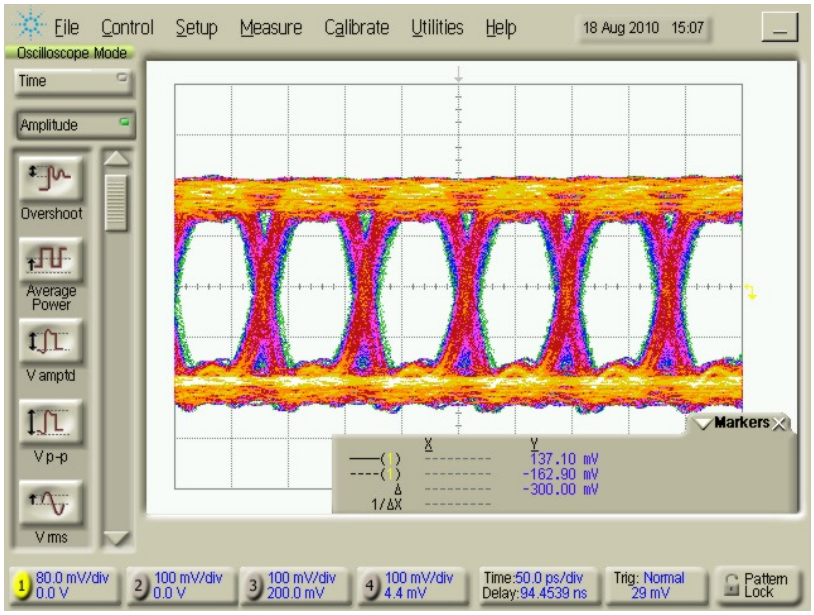


Figure 8.14 12 Gbps output externally buffered and set for about 300 mV amplitude.

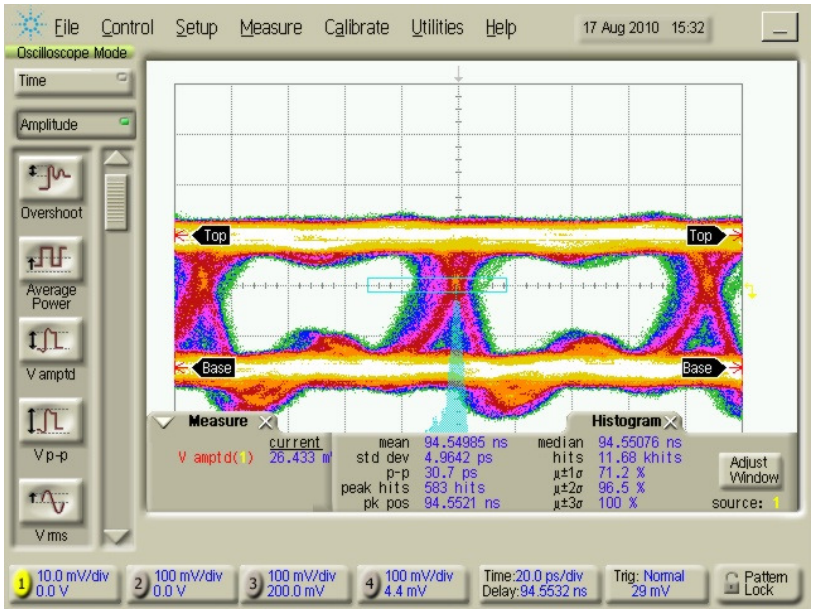


Figure 8.15 12 Gbps output externally buffered and set for about 26 mV amplitude.

In order to separate the RJ and DJ jitter components, the characterization tests are performed in two manners. In one case we use a pseudo-random serial bit pattern to obtain the traditional data "eye" as illustrated so far in this paper. However, the jitter histograms obtained include both random and deterministic effects. To isolate the

random effects, we change the test pattern to a repeating/cycling pattern wherein the data does not change during the jitter-histogram measurement, i.e. the pattern is "stable." The jitter histogram is used to analyze a single (repeating) edge as shown in Figure 8.16. Using a comparable-length measurement (10 kHits in this example) to that of the data-eye pattern, we measure the total jitter (peak-to-peak) for this new signal. Since the data is not changing, this signal contains almost entirely random-jitter characteristics. So we can subtract this "random" value from the data-eye total-jitter, to obtain an estimate for the DJ portion. The standard-deviation of the random-jitter measurement provides a close approximation for RJ. As described earlier, the two values (DJ and RJ) can be used to predict the jitter characteristics for different bit-error rates (TJ = 14.1xRJ + DJ at BER=10^{-12} for example). (cut from previous section with the eye figure 8.9) It also includes jitter from the clock reference and the measurement instruments. In more detailed measurements (see Figure 8.16) the RJ and DJ components were separated. For this signal RJ=1.3 ps (RMS), and DJ=14.5 ps (including measurement instruments). It is important to separate the RJ and DJ contributions because we can then use a simple extrapolation to predict the data eye opening at a BER of 10^{-12}. For this we use TJ = 14.1xRJ + DJ. When applied to this signal, the total jitter at 10^{-12} BER is 28.6 ps. Therefore the data eye opening is 66% UI (unit interval). One UI is 83.3 ps at 12 Gbps.

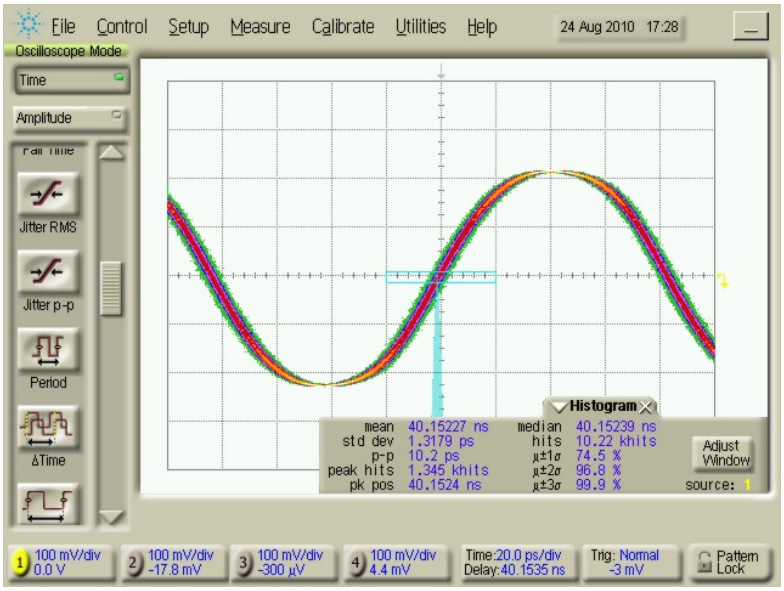


Figure 8.16 Single edge measurement.

One additional feature of the 12 Gbps module is the ability to multiplex TWO 12-Gbps signals and form a 24 Gbps serial test pattern. As with the 2.5 Gbps dual-channel modules, two serial streams at an identical rate but offset by half a bit can be combined using a very high-performance, such as InP or SiGe, exclusive-OR gate. An example 24 Gbps data eye generated using this methodology is shown in Figure 8.17. Here the peak-to-peak total jitter is measures as 18ps on a 10ps per division scale. The example is shown using over 40 kHits to illustrate the signal stability and relatively-open "eye". Keep in mind that this circuit was built using all off-the-shelf components for a cost considerably less than that of commercial instruments. Also note (again) that all the measurements shown include measurement errors inherent in clock-reference source (about 1 ps RMS), and the sampling oscilloscope (about 200 fs RMS).

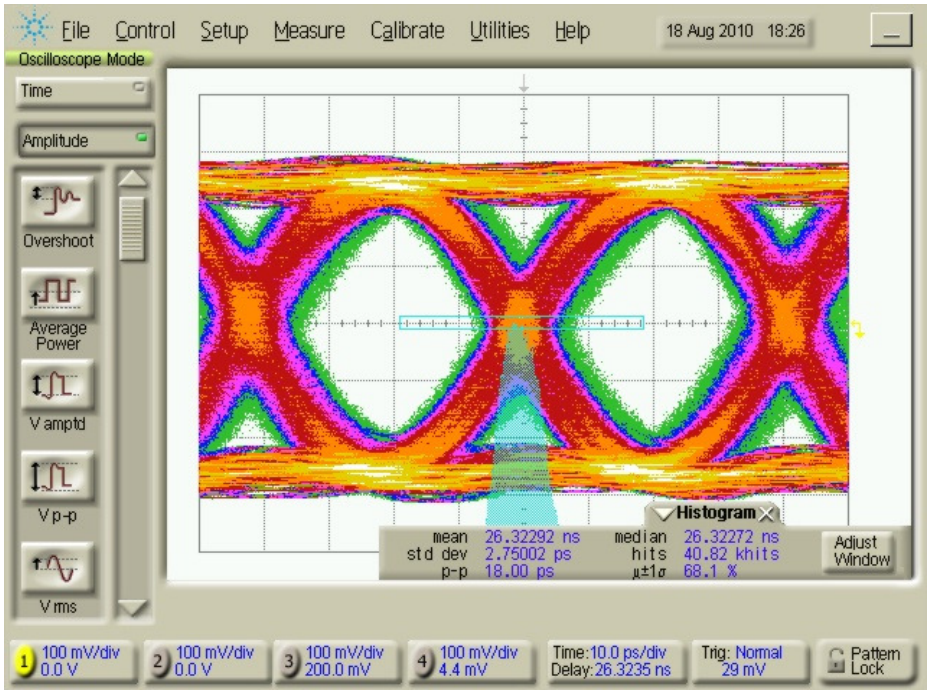


Figure 8.17 Multiplexed TX module performance at 24Gbps.

8.3 Burst mode receiver

Compared to parallel interfaces, serial interconnections lead to smaller design implementations due to fewer signal lines, lower power dissipation, and higher bandwidth opportunities. These higher bandwidth capabilities are contributable in part due to the simplified design requirements and lower crosstalk across the reduced number of signals compared to a parallel implementation using more signals at a slower individual rate. However a more a significant source of improvement is from eliminating the clocking relationships between wide collections of signals relative to a single clock. Many serial implementations eliminate the need for an accompanying clock as an independent signal, embedding this information as part of the serial stream of data. This however does introduce a new design constraint in the necessity for a circuit to recover this embedded clocking information at the destination which can in turn be used to sample the received data.

The most commonly used solution for this problem is employing a phase-locked loop in some way to align a local clock reference to the embedded clocking information within the serial stream. The specifics vary by implementation, but this reference is usually edge transitions within the received signal though some alternative approaches have been explored [75]. Establishing the phase relationship to the clocking information is, in most conventional clock and data recovery (CDR) implementations, a non-instantaneous operation. Accordingly, using these edge transitions from within the data stream is problematic under bursty [76] data conditions where this relationship may be varying faster than the CDR circuit can maintain alignment. This is exactly the situation which occurs within the Data Vortex and is the primary reason why a parallel clock is transmitted within packets when using the 2.5 Gbps transmitter modules. Unfortunately, as mentioned at the end of Chapter 6, this methodology and the corresponding physical devices cannot really be utilized to push the system performance to higher signaling rates.

Transmitters capable of 12 Gbps were presented in the previous section, but the corresponding receivers are not directly compatible with the bursty signals associated with the Data Vortex because of the implementation of CDR logic. Similar networks employing packet switching [77] and short burst communications [78] are not constrained by these asynchronous arrival concerns as the switching fabrics employed are of a fixed and consistent depth. This allows for a simplified test and data recovery implementation as a synchronous timing relationship can be established between the source and receiver nodes in the native operational modes, similar to the physical layer verification process used for the Data Vortex. Alternative solutions suitable for short-burst and asynchronous applications have been explored to reduce the overall synchronization time at lower signaling rates such as 1.25-2.5 Gbps [79,80] by employing burst-mode receivers to improve the settling time of the optical to electrical conversion elements. Higher-performance integrated solutions for a full CDR implementation, which are necessary for full-speed functional verification of these high-

performance systems targeted at 10 Gbps per channel applications are also in development [81,82,76].

A burst mode receiver, capable of operating at 10 Gbps and locking to an incoming data stream almost instantaneously, was developed as part of this research. A high-level view of the receiver logic is shown in Fig.3. The 10 Gbps incoming data is input to the block labeled "1:4 CDR Prescaler." The Frame signal, recovered from an incoming packet, is also used to prime and enable the circuitry. The Frame signal 'arms' a tuned-oscillator which is triggered by the first edge of the payload data. Without this framing signal, the CDR circuit will be inactive and ignore any input on the high-speed data-in port. The "Delay Adjust" signals are used to define the relative phase between the recovered clock and the payload data. This is set during calibration of the module to account for internal delays, and can also be used to sweep the clock phase during testing.

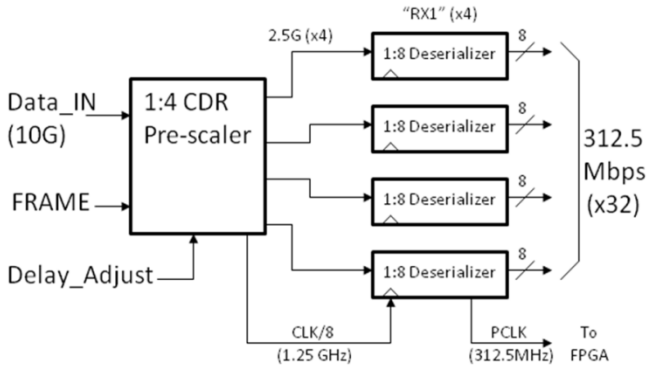


Figure 8.18 10 Gbps data recover – top-level logic.

The 1:4 CDR Prescaler accomplishes two tasks in parallel: (1) it recovers a 2.5 GHz clock (or, to be more precise, it synthesizes a clock locked in phase to the reserved CDR bit), and (2) it performs a 1:4 demultiplex of the 10Gbps data into four 2.5 Gbps signals. These four signals can each be further deserialized by the burst-capable 2.5 Gbps receiver modules presented earlier.

The architecture of the 1:4 CDR Prescaler is shown in Fig.4, and timing diagram for its operation is shown in Fig.5. Here the incoming 10Gbps data is brought into a fanout block that creates two identical copies with a fixed delay relationship. In the figure, the top path is used for 1:4 deserializing while the bottom path is used for clock-recovery. Deserializaiton (top path) is accomplished by creating four copies of the data, adjusting their delays by increments of 100 ps, and clocking into a 4-bit register. This path includes a "Coarse Delay" block that provides rough alignment of these data to the recovered clock signals. Delay increments are accomplished using PCB transmission lines.

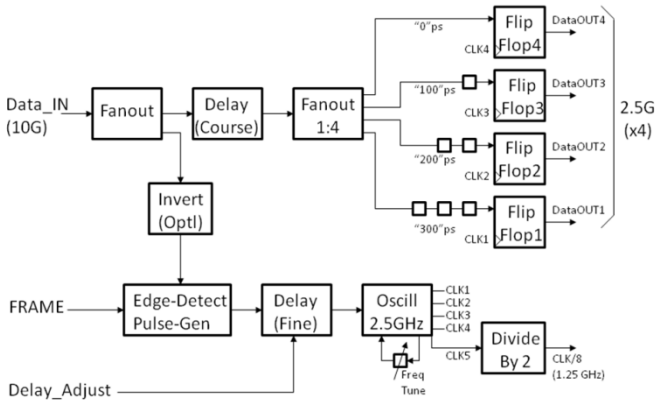


Figure 8.19 1:4 CDR/Prescaler logic..

The lower path feeds a copy of the 10G data to the "Edge-Detect Pulse-Gen" block. This block is first "armed" by the "FRAME" signal so that it is ready to produce a pulse upon trigger from the first rising edge of the data (the reserved CDR bit). An optional inverter block allows us to select the first FALLING edge as an alternative solution.

The pulse is transmitted through a programmable delay circuit (Fine Delay). Several control signals in the "Delay_Adjust" bus are used to define the fine-delay value. In normal (operational) mode, the delay remains fixed with the recovered clock phase aligned to the middle of the 10G data-eye, as received by the desierialzer flip-flops.

However, for characterization testing this fine delay can be swept across several bit periods.

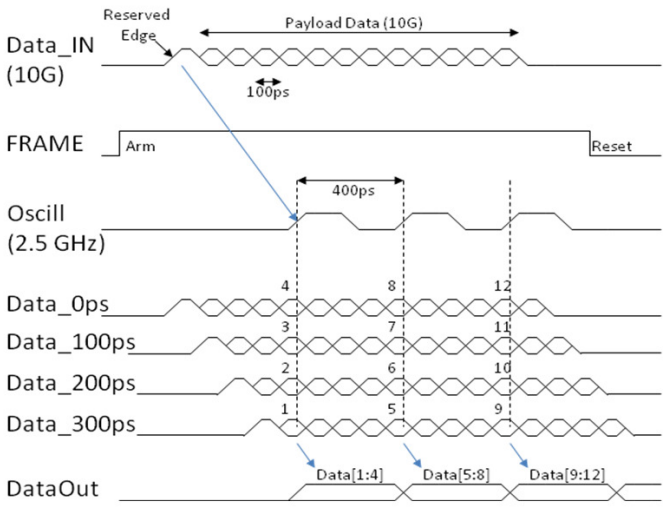


Figure 8.20 1:4 CDR timing diagram (simplified).

The 2.5 GHz Oscillator is an edge-triggered, tuned circuit that produces the clock signals needed for the four deserializer flip-flops. This circuit is made up of a very high-speed (SiGe) gated "ring" oscillator. At the beginning of the packet, the FRAME signal is used to "ARM" the edge-detector circuit, as described above. The edge-detector generates a single 0-to-1 logic transition that is triggered by the first (reserved) edge of the payload data (as in Fig.5). This transition is tightly-synchronized to the payload data and is a nearly-ideal timing reference for initiating the start of the ring oscillator. The frequency of the oscillator depends upon its SiGe gate delay and the delay of a short segment of transmission line which is tuned to obtain a characteristic oscillation frequency of 2.5 GHz. After synchronizing to the initial edge, the oscillator free-runs during the short (~12ns) time of the payload data. It will continue to run until the Frame signal is de-asserted and returns to logic zero at which point the oscillations stop.. Unlike conventional phase-locking approaches, there is no feedback or further realignment of the clock to the data. Therefore it is critical that both the oscillator and the payload data maintain a constant rate during the short time of the burst. A low-jitter, high-speed SiGe

1:4 fanout buffer is used to obtain the 4 synchronous clocks used by the deserializer flip-flops.

The prototype is successful at meeting these challenges. In Fig.9 the output of the oscillator is shown on a relatively coarse time scale. Here the entire burst is visible, lasting a bit over 16ns. For this demonstration there are 40 complete clocks generated (plus one or two more, depending upon the trailing edge of the FRAME which are NOT used in subsequent logic).

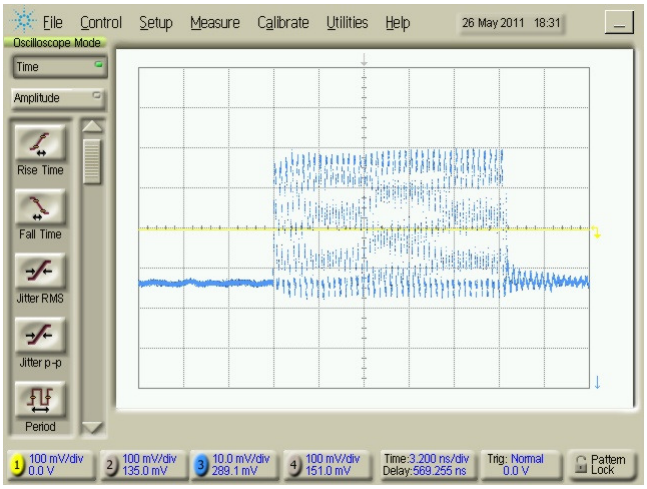


Figure 8.21 Recovered clock oscillator output – entire burst.

A bit more detail is visible in Fig.10 which shows the first several clocks on a finer time scale. The first cycle shows a very small (~8ps) timing error (pulse shorting), with an initial period of 393ps. The nominal clock period was 401.5ps, and had only 1 or 2 ps of cycle-to-cycle variation after the very first cycle. In practice we would more tightly-tune the oscillator to get closer to the ideal 400ps period. This initial tuning was sufficient for checking the circuit functional behavior.

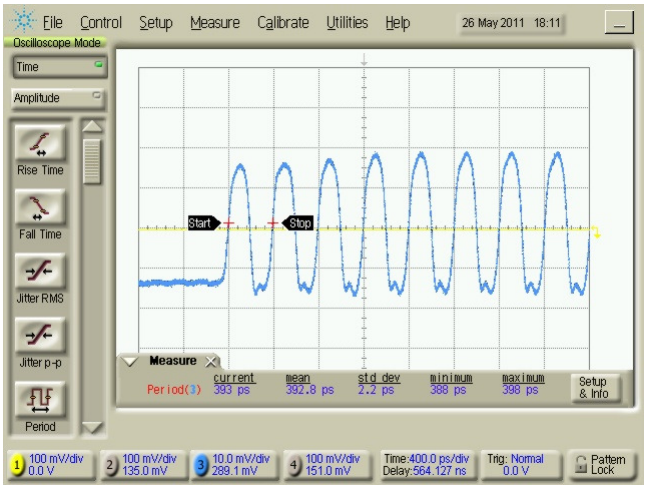


Figure 8.22 Recovered clock – details of the first several clock cycles.

Of particular interest was the very first cycle, as it shows ability of the oscillator to handle a "cold start" from a long period of idle activity. In practice, the arrival of packets may occur every 25.6ns, or there could be arbitrarily long delays between packet arrivals. In this demonstration one of 16 packets contained payload data, with 15 dead cycles (15x25.6ns of inactivity). The very first cycle of the recovered clock is shown in Fig.11. This initial period is measured to be 392.79ps (i.e. about 8ps less than the typical, and about 7ps less than ideal). The second cycle showed less error (2nd period = 399ps), and all subsequent cycles were typically 401.5ps. Jitter was measured as 2.2ps rms, which was only slightly more than the 1.4ps measured for the payload data signal.

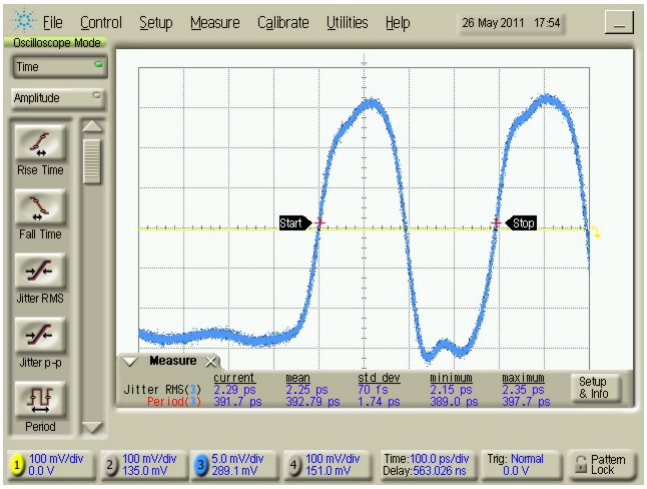


Figure 8.23 Burst-mode recovered clock – first cycle.

The overall output of the burst receiver is shown in Fig. 13. The top most signal is the clock generated by the oscillator (time shifted relative to the data due to a difference in cabling) while the remaining signals are the four demultiplexed serial streams. The demultiplexed streams are at a 2.5Gbps data rate and the bit width is correspondingly 400ps. Though only the first ten bits of the individual demultiplexed sequences are shown on screen, the clock is sufficiently stable to recover the maximum packet payload of 19.2ns worth of data. The high-speed signal used for test purposes was a 16-bit repeating pattern so the demultiplexed signals repeat every four bits.

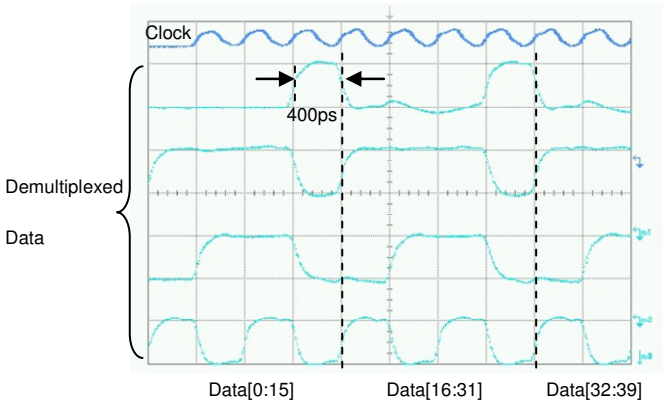


Figure 8.24 Recovered clock and 1:4 demultiplexed data.

8.4 Memory extension module

While the majority of the application modules discussed so far are related to improving the transmission or capture capability of the underlying system, the data lines connected to the module slots are connected to general purpose I/O on the FPGA and are available for any conceivable purpose. Additionally there is nothing prohibiting multiple module slots from being bridged to allow for the signals from multiple slots to be combined if the pins specific to any one slot are insufficient to meet a set of design requirements. One such module was designed, visible in the bottom left of Figure 8.1, to support access to a DDR2 memory DIMM. This module was intended to expand the total amount of memory storage space available to the pattern generators or to allow a larger memory region to be used for system level emulation scenarios like that presented in Section 7.5.2. The module was designed and fabricated but not fully integrated into the test system.

8.5 Module Combinations

Given the new module designs described in this chapter, it is possible to envision several additional test configurations utilizing the test platform bank A resources that combine the module features in different ways. Some of these are listed below:

- 2.5 Gbps – 16 transmitter channels
- Multiplexed 5 Gbps – 8 transmitter channels
- 2.5 Gbps (x8) with DDR memory module
- 2.5 Gbps – 8 bi-directional channels with loopback
- 12 Gbps – 4 bi-directional channels
- 20 Gbps – 2 bi-directional channels

In the first listed configuration, all of the transmitter modules are upgraded from the original single-channel to dual-channel 2.5/5Gbps versions as shown in Figure 8.25.

This provides twice the number of outgoing high-speed signals (16 instead of 8), and therefore twice the aggregate data-rate (now 16x2.5 Gbps=20Gbps). Of course, applying this many new channels to the Data Vortex would also require twice the number of optoelectric components to handle the increased channel load, but would also allow for higher utilization of the single optical fiber available bandwidth. Alternatively, since the receiver resources populated in Bank B remain unchanged, these additional channels can be combined with unused Header/Frame channels from the development platform to create additional. These additional injection nodes can be used to evaluate the network under load from multiple injection points without requiring additional development platforms.

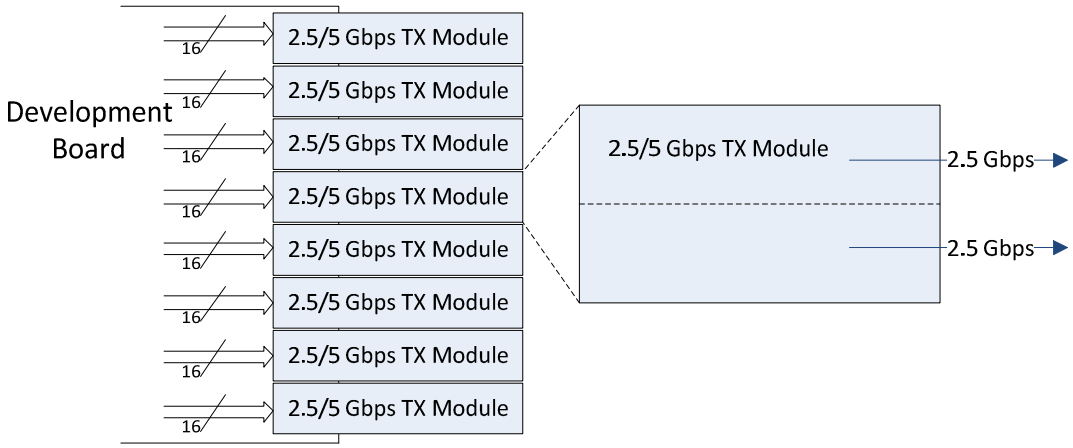


Figure 8.25 2.5 Gbps x 16 configuration.

In the second listed configuration, all of the original modules are replaced with the new dual-channel versions as above. However, in this configuration these modules operate in the multiplexed mode at 5 Gbps as shown in Figure 8.26. This provides the same data rate as the first new configuration (20 Gbps in each direction), but has the added advantage of requiring only 8 E/O and 8 O/E modules (only half of that required by the first configuration). The trade-off is that this second configuration requires a higher degree of timing accuracy, in order to support parallel synchronization of 8x5

Gbps signals, and would require complementary capture modules at a similar signaling rate.

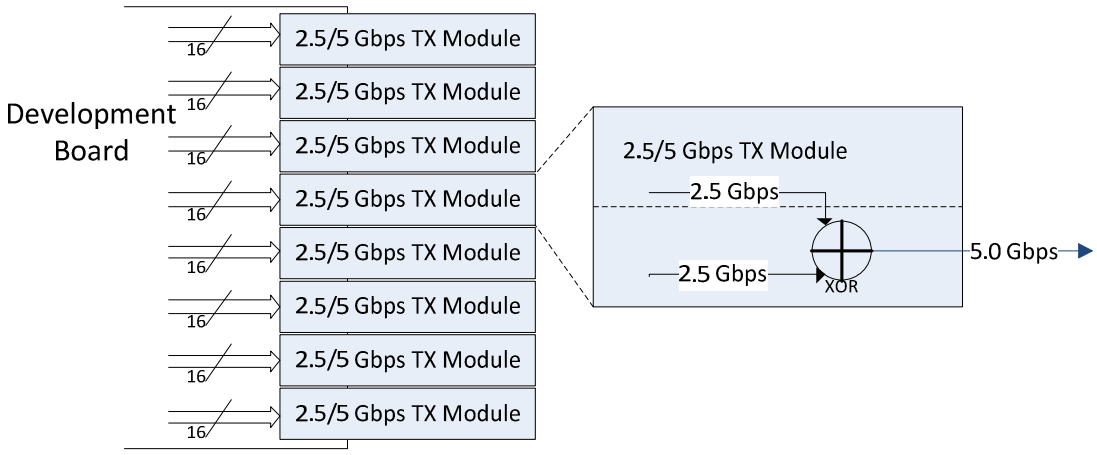


Figure 8.26 5.0 Gbps x8 configuration.

In the third configuration, four dual-channel TX modules can be used to obtain 8 transmission channels which is equivalent to the original system. While there is no direct gain in aggregate or individual signaling rates, this frees-up four Bank A slots which are then used to support a high-speed connection to a removable memory DIMM. Such a memory module has been designed, fabricated, and FPGA firmware support is being developed. This memory can serve a variety of options, including but not limited to storing test patterns in the absence of a host computer or enabling the Development Platform to operate as an autonomous memory node attached to the network. Such a memory node would enable higher-level emulation of low-latency, shared-memory access amongst processors (real or virtual as emulated by test platforms) connected to the Data Vortex optical network.

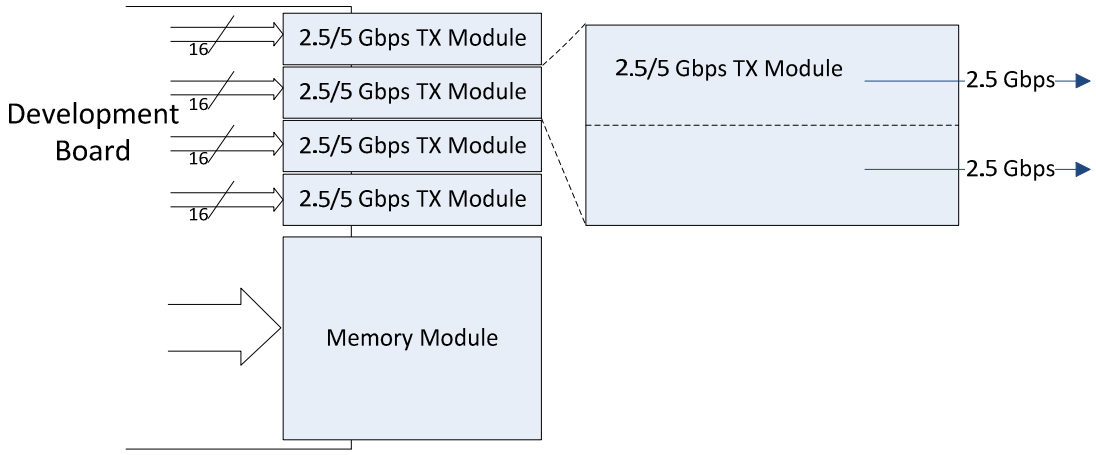


Figure 8.27 2.5 Gbps x8 with local memory.

By pairing-up dual-channel TX and RX modules in alternating slots, the loopback provisions of these modules can be exploited as shown in Figure 8.28. The signals as generated by the test platform can be applied to the DUT and the corresponding DUT output signals returned to the tester for evaluation or the generated signals can be locally looped back upon the tester for self-test and/or calibration purposes. The final option is to configure the relays to enable the DUT to drive itself in loopback, bypassing the tester almost completely.

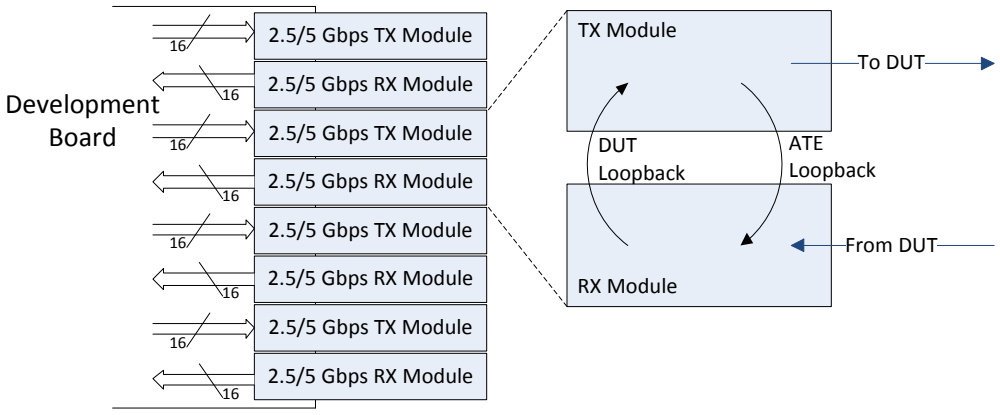


Figure 8.28 TX and RX modules with loopback capability.

In the 5th and 6th listed configurations, the 2.5/5 Gbps dual channel modules are replaced with the newer TX/RX 12Gbps modules. The TX modules can operate in either non-multiplex or multiplexed modes (Figure 8.29). With four each of the transmitter and receiver modules, an aggregate bandwidth of 48Gbps is possible for bi-directional operation. Using eight transmitter modules, rates of almost 100Gbps aggregate is achievable in the form of 8x12Gbps or, in the multiplexed mode, 4 channels at 24Gbps.

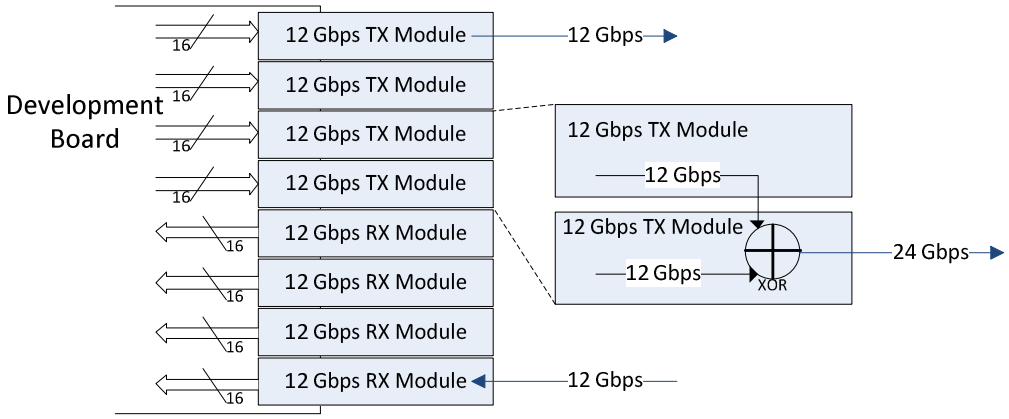


Figure 8.29 TX and RX modules with 12/24 Gbps capability.

8.6 Routing expansion board

It was recognized that one of the injection modes that needed to be more thoroughly studied on the Data Vortex was simultaneous injections from multiple ports, in fixed or random patterns. It is possible to define the payload of a packet with two high-speed channels, using a single channel from each logical injector to be modulated onto a common wavelength to serve as a clock for recovery purposes and the second as an identifiable and unique 'marker' to track the packet origin. In this way up to four packets can be constructed using eight of the nine Bank A slots as data transmitters. However, each generated packet requires three to five, for small network topologies and depending on the particular routing options available, unique Frame and Routing signals. A very rudimentary experiment was constructed using this principle, allocating the signal

generation resources (Figure 8.30 and Figure 8.31) of a single test platform to create three 'logical' injection sources.

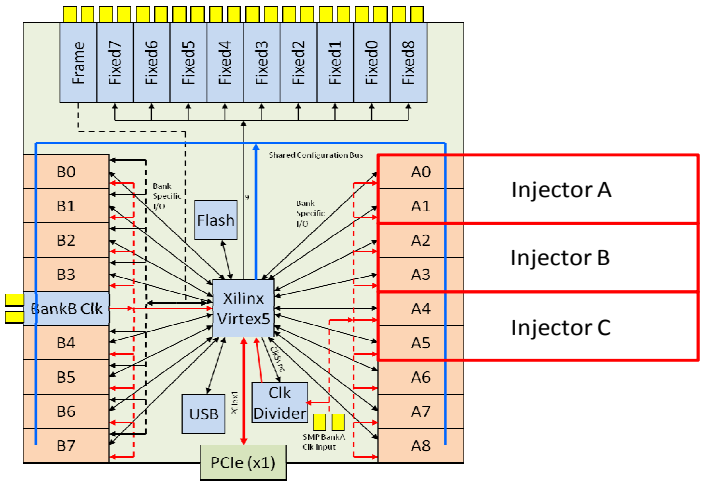


Figure 8.30 Payload injector allocation.

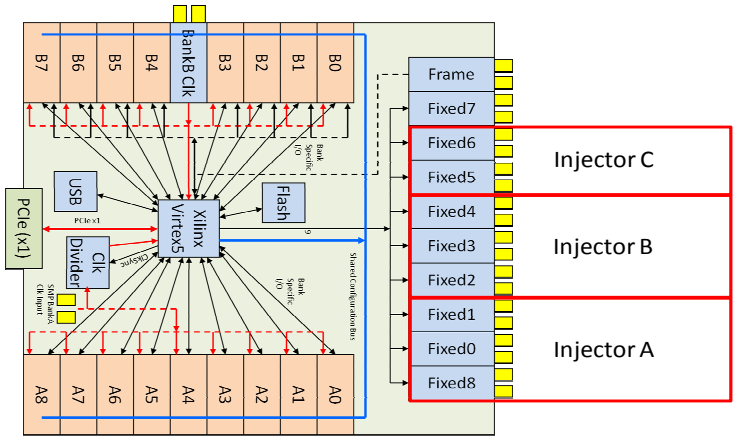


Figure 8.31 Frame and routing signal allocation.

A single reception node had to be used as part of the test (Figure 8.32) due to the requirement that all receiver modules on a platform use a single receiver clock. Using this setup, potential collisions could be prepared for injection into the network and the results, monitored by payload wavelength and by extension the receiver channel on the test platform, observed through the system.

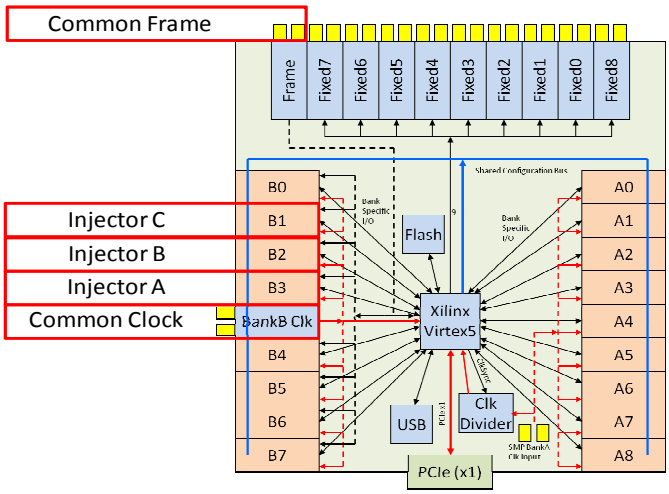


Figure 8.32 Receiver signal allocation.

The limiting constraint to the deployment of this test solution is the availability of Frame and Routing signals. At one Frame and four Routing bits per injection node, for the four sets of payload channels that can be achieved with the Bank A resources, 20 Frame-formatted signals would be required for a fully addressable 36 node network. If signal capture is ignored and all nine payload channels are used for 'marker' purposes only as many as 45 of these Frame-formatted signals would be required.

While these additional routing bits can be logically generated by the FPGA, the difficulty is in electrically presenting them to the DUT. To this end, a routing expansion board was proposed to be compatible with the Bank A module slots. Of the 18 dedicated data lines to these modules, 8 can be used as directly modulated signals from the FPGA into delay circuitry, identical to the generation circuits at the top edge of the test platform. The remaining signals would be required to address the delay circuitry to ensure that each of the additional signals added can be skewed in the same fashion as the other signals implemented in the system. This is especially important because the phase of the signals added by this module may be significantly out of phase to the equivalent signals on the test platform. One or more of this module could be implemented to add routing bits to the system as needed. This module is currently in the design and development stage.

CHAPTER 9

SUMMARY AND CONCLUSIONS

9.1 Summary

The objective of this research was to develop an FPGA based architecture to provide test capability for multi-Gbps source-synchronous devices while natively supporting the signaling protocol requirements of the device. In Chapter 1 an introduction to the motivation of this work was provided. A more extensive overview of test methodologies, with respect to the historical development of test systems in general as well as recent innovations changing the field, was presented in Chapter 2. These methodologies have provided a combination of motivation, inspiration, and in some cases the initial foundation for some of the expanded designs and implementations developed in this research.

Chapter 3 described the Data Vortex optical packet switching network whose test and evaluation requirements has served both as the original motivation for this work and a demonstration platform for the performance and capabilities of the resulting testing architecture.

Some stages of the design and evolution of the Test Development Platform, including previous and current designs, were presented in Chapter 4. The separation of the FPGA in a Digital Logic Core from the application logic, moved to removable modules, allowed for the exploitation of both the flexibility and versatility of the FPGA while enabling targeted use of application modules for specific test implementations. The test platform incorporates, in addition to the titular FPGA of this work, a combination of support infrastructure for the application modules, limited direct signaling capability, and communications interfaces to connect to an external computer system for additional control, configuration, and processing capabilities.

Some application modules and the respective signal generation and capture responsibilities, especially as they pertain to the Data Vortex, were presented in Chapters 5 and 6. Chapter 5 focused on the generation and transmission of signals, with a particular focus on leveraging signal serialization, time shift functionality, and variable amplitude buffer drivers to create source-synchronous, packetized signal bursts compatible with the Data Vortex signaling protocols. Chapter 6 focused on the capture and recovery of these source-synchronous packets, fundamentally incompatible with most test solutions, into a format compatible with the tester FPGA. Combined with the test development platform, these modules enable the creation of a high-performance multi-channel test system with each channel operating in the multi-GHz range. This is especially demonstrated through the application and recovery of eight 2.5 Gbps data channels, in addition to the packetized clock, framing, and routing signals required for the interface protocol, through the Data Vortex.

The processing and analysis capabilities of the test system are described in Chapter 7. Simple test evaluation can be performed as with standard functional testers or advanced processing and data manipulation, intended to more easily interface the system to protocol reliant and non-deterministic systems, can be utilized. This style of processing is enabled in this architecture due to the shared and universal access of the data across pins and signaling channels rather than a narrow pin-by-pin implementation. Data can be acted upon at the system level, allowing the tester to dynamically react to the ongoing state of the test or to adapt to situations such as data alignment skew which would be difficult or impossible in other test solutions.

The design, creation, and performance of additional modules and the test application variations they can support were presented in Chapter 8. Some of these modules have been designed to allow the system to operate with additional high-performance signals or increased performance on the same number of channels. Some of the other modules presented enable alternate test configurations by re-tasking modules

which would be otherwise be used for high-speed signal generation or capture and using them instead to create many more lower-speed signals or to support general purpose functionality such as added memory storage.

9.2 Contributions

The summary given above highlights the achievements of this research. Based upon these achievements, the major contributions of this thesis are:

9.2.1 Test development platform

An FPGA based architecture for the creation of test systems is described in this thesis. The core element of this architecture is the test development platform which incorporates the central FPGA and is described in Chapter 4. The physical separation of the FPGA, communications interfaces, and support infrastructure implemented on this test development platform from the application specific logic allows for easy upgradability of the independent elements to exploit next generation devices as they become available.

The test platform has been designed to support many application modules, in quantity and variety, allowing for the creation of multi-channel systems adaptable to specific testing needs. Existing modules can be added and rearranged or new ones developed while reusing most or all of the underlying platform support capabilities and software. Alternatively, a new test platform can be constructed using a newer or larger FPGA, an FPGA from a different manufacturer, or implementing a different control interface without losing support for the existing inventory of application modules.

9.2.2 High-performance application modules

While FPGAs are powerful and flexible, they are performance limited with respect to the low jitter and high-speed signal requirements for high-performance systems like the Data Vortex. Chapters 5 and 6 describe the creation and performance of application modules, driven and controlled by the FPGA, capable of generating and capturing these high-performance signals which would otherwise be impossible utilizing the FPGA resources alone. These modules incorporate basic design features such as signal amplitude control and timing adjustment essential to supporting accurate and quality high-speed signals.

Additional modules, incorporating advanced features such as multi-channel multiplexing, loopback signaling paths, and even higher signaling rates, were described and demonstrated in Chapter 8. These modules display the versatility and expandability of this joint system for applications outside the original scope and definitions of the target test bench.

9.2.3 Protocol and application specific dynamic test

In conjunction with the hardware elements above, software on a host computer and/or firmware logic within the test platform FPGA can be utilized to generate stimulus signals and process responses appropriate to evaluate the device or system under test. A combined testing approach, capable of not just replicating a functional equivalent interface but instead creating an emulation of the system interface, is presented in Chapter 7. Test routines or sequences can be initiated by a user or be programmatically reactive to the ongoing state of the state of the system.

9.2.4 Cost-efficient customized test implementations

This thesis presents a design approach that allows for cost-efficient implementation of customized test solutions through the development of application modules targeted at the desired test capabilities. The existing test platform can be reused with a variety of application modules, as demonstrated in Chapter 8. New modules can be implemented using the universal plug-in footprint and integrated into the test architecture in a time and resource efficient fashion.

9.2.5 Data transport and control

A data transport and control solution for this test architecture is presented in this thesis. The solution is comprised of three parts, presented in Chapter 4, including a memory topology within the FPGA, a software interface on an external computer system, and a signaling interface that can be interfaced to the FPGA such as USB. The memory topology created is essential for providing external access to memory resources within the FPGA as well as facilitating data transport within the FPGA across logic regions associated with differing clock domains. Software on a host computer, interfacing to the test system over a non-proprietary high-level interface can be tailored to the tester requirements. The software can be used for low-level bit accurate access to memory components or high-level abstraction, insulating the user from the bit-level implementation details of the underlying system. The control interface also allows for user designed and initiated functions such as timing updates, system reset, or on-demand test executions through a simple instruction execution system.

9.3 Conclusions

The objective of this research was to develop an adaptable test architecture appropriate to test high-speed, source-synchronous protocol interfaces. This was achieved through the design and creation of a combination of an FPGA-based test interface board and application modules. The resulting system was verified through successful application of a variety of tests to the Data Vortex optical packet switching network, though the underlying system and physical implementation is applicable to a wider range of devices and systems.

The methods presented in this thesis offer five distinct contributions presented in the previous section. These contributions will ensure that this architecture, or individual components of it, will be applicable for research into future test solutions as newer technologies and design approaches become available. Though the original design was created with the intent of creating a final test system supporting eight channels of 2.5 Gbps signals with a total throughput of 20 Gbps, additional transmitters utilizing some of these newer technologies and design approaches have already been implemented pushing that amount to double and beyond.

9.4 Future Work

Due to the modular nature of this work, many avenues for future development are available. Chapter 8 includes a collection of modules that are compatible with this system, but there are many more possible combinations and applications not presented. The existing modules can also be redesigned if newer and better performing devices become available to improve the overall system performance, such as higher quality serializer/deserializer devices, improved timing accuracy delay buffers, or additional DC or loopback functionality is required.

Also, the present command instruction infrastructure within the FPGA is implemented as a very simple state machine. The state machine observes a particular address, executes the indicated instruction when the address has a non-zero value, and clears the value when execution is completely. There are some operations which should logically remain independent, including synchronization pulsing, LFSR reset, and digital clock manager reset to name a few developed as part of the research. These functions, in the present design, are executed by the software and triggered by the user having pressed a button which executes each operation in turn sequentially. Once an instruction is initiated, the ongoing execution must complete before a new instruction can be presented, which requires polling of that memory address and waiting till the value is zeroed out before presenting a new operation. There is nothing fundamentally wrong with this implementation, but it can be improved upon. Each operation can be defined as a pseudo-instruction in a very simple processor architecture supporting these pseudo-instructions as well as more standard operations. This would enable the creation of actual test programs, including load, stores, mathematical functions and conditional processing to be constructed an executed.

And finally, as general and FPGA device technologies continue to develop, the base system platform can be improved to take advantage of any new features or improved capability which may become available. The Virtex5 is now two generations old relative to the newest 7 series products becoming available from Xilinx. The use of a newer, higher performance device could improve core performance or a simpler device could reduce the system complexity and cost. Exploratory efforts to migrate elements of this architecture to a Spartan6 FPGA-based board. The benefit is that the device has a lower cost with comparable overall performance with regards to the programmable logic, internal memory, and general purpose I/O. The device does not support RocketIO so there is no PCIe. However, the prototype has also allowed for initial tests utilizing a new USB physical interface chip, the Silicon Labs device mentioned in Section 4.5.3, while

maintaining compatibility with the underlying memory topology and by extension all related functionality.

APPENDIX A

PHYSICAL BOARD DESIGN AND LAYOUT

In this appendix, details of the test platform's logical design and physical layout are presented. The test platform is designed using the Mentor Graphics PADS software suite. Individual components ranging from devices such as the FPGA, USB microcontroller, and clock distribution buffers to physical connectors such as those used for power, high-speed SMAs, or the multi-pin connectors for the expansion modules are implemented in a library common to all elements of the design suite. The logical design is constructed in schematic form from this component library using PADS Logic. Design elements are instanced from the library and are connected pin-to-pin with other elements or to system wide structures such as power or ground nets. The physical layout is constructed from that schematic in PADS Layout. Each library element has a corresponding physical 'footprint' that incorporates the copper pads and other physical characteristics required to electrically connect and mount the component to a printed circuit board (PCB). The software packages can be used in conjunction to guide the user through the physical placement and routing process while ensuring that only the desired elements are connected to each other.

Physical Layout:

The images below show the structure of the PCB, one copper layer at a time. The top layer silkscreen outlines and labels are present in all images to help provide a consistent frame of reference.

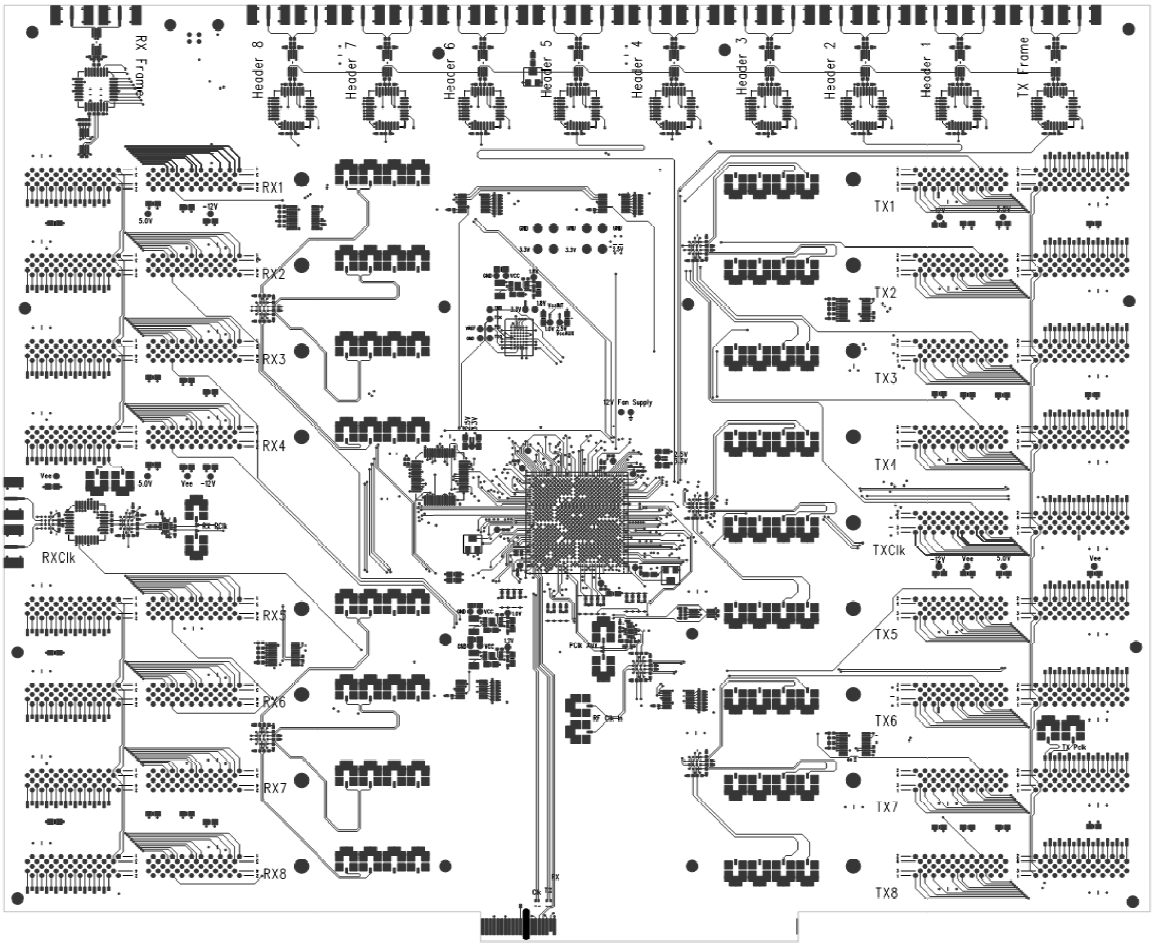


Figure A.1 Top layer

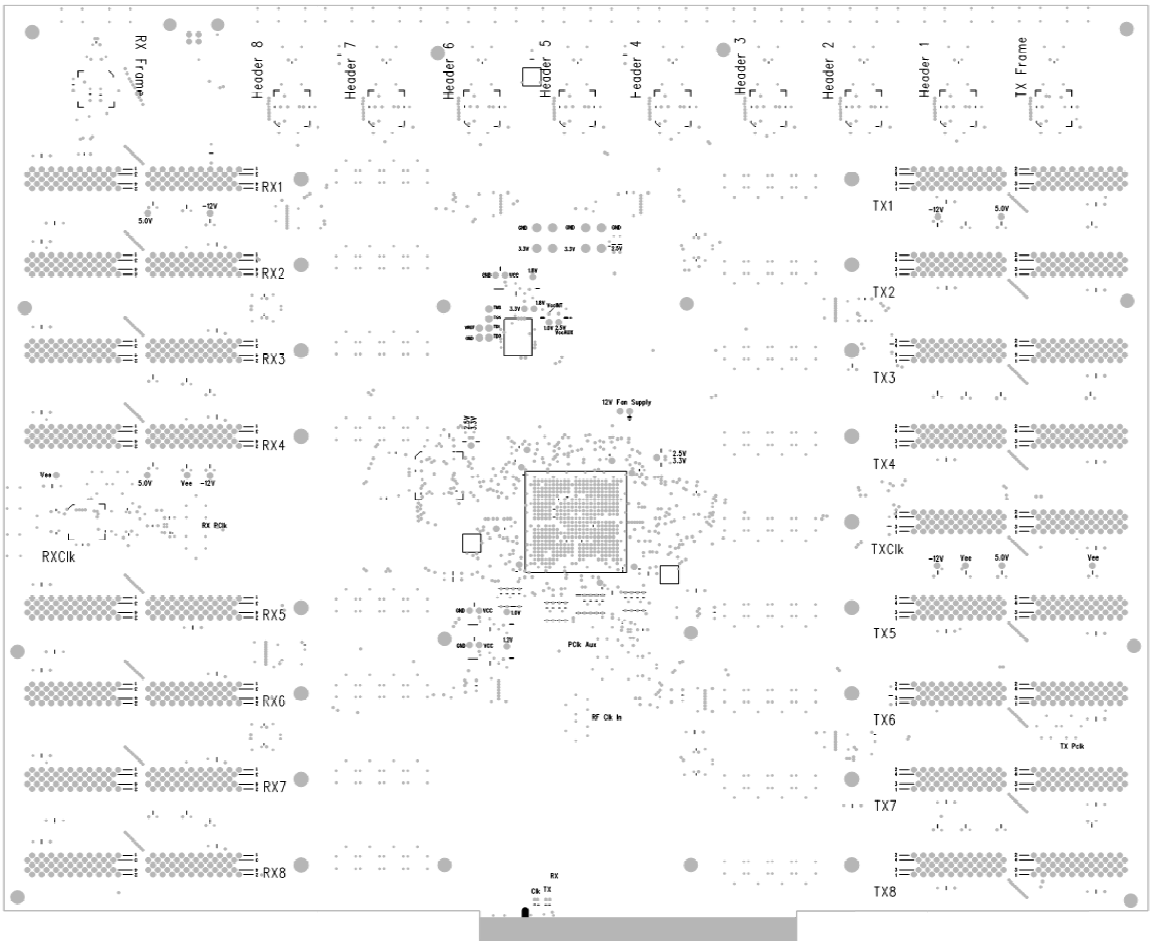


Figure A.2 Ground plane - layers 2, 4, 7, and 9

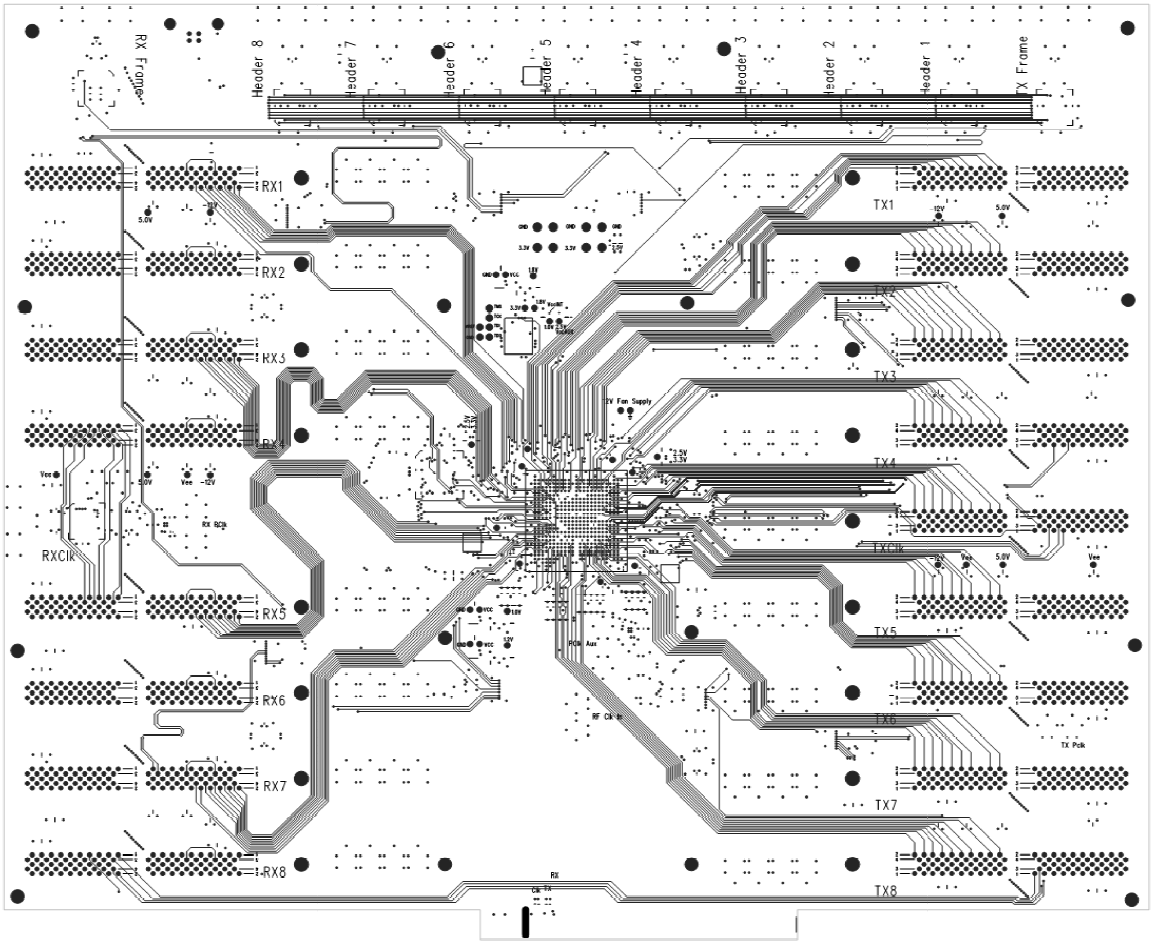


Figure A.3 Inner 1

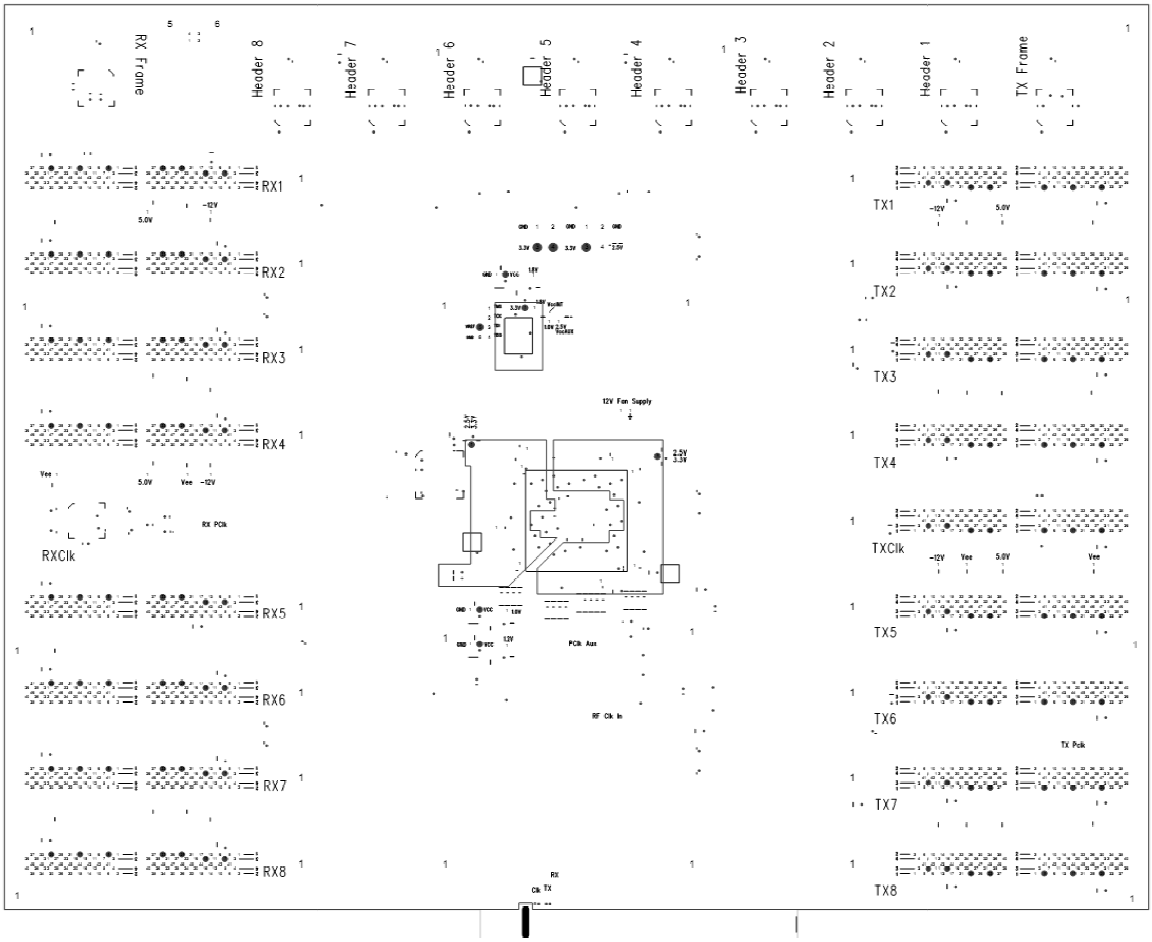


Figure A.4 Power 1

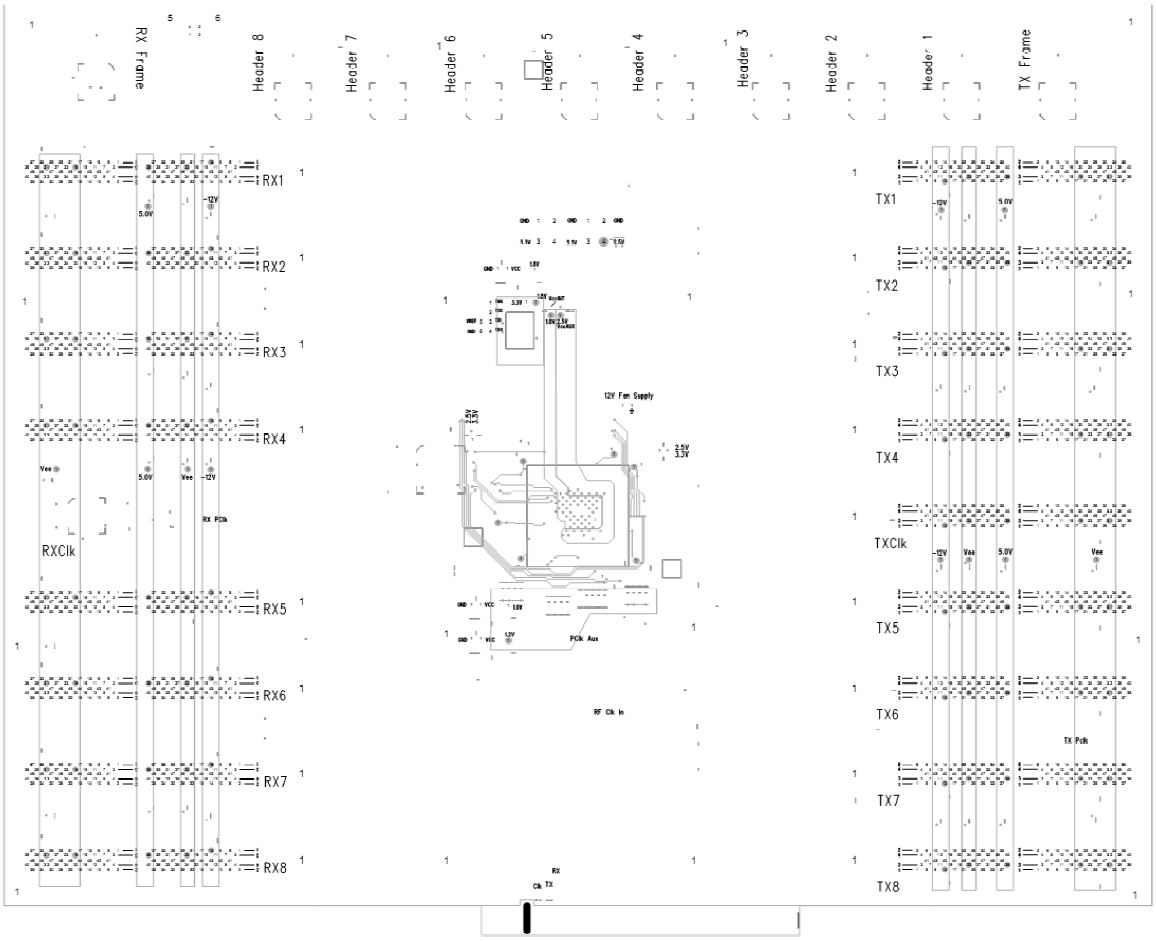


Figure A.5 Power 2

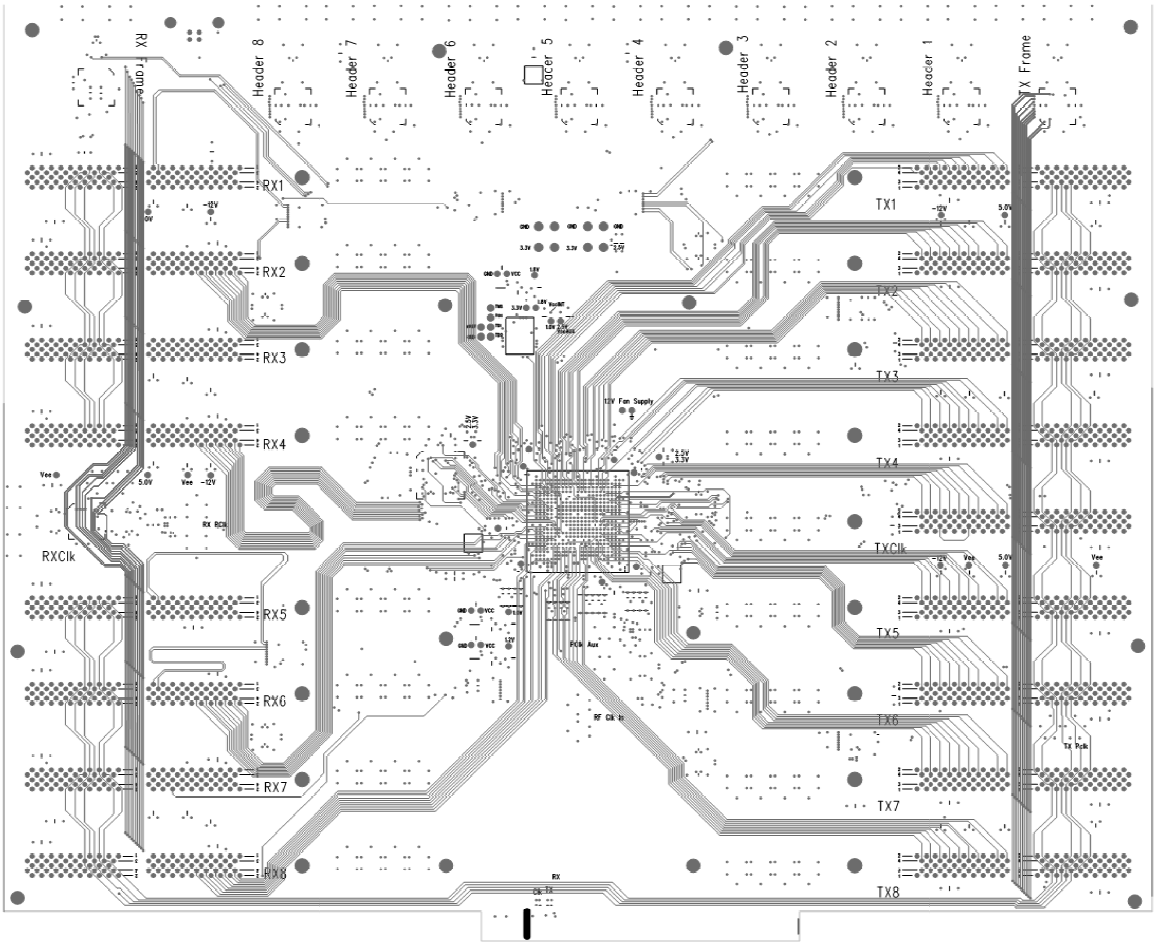


Figure A.6 Inner 2

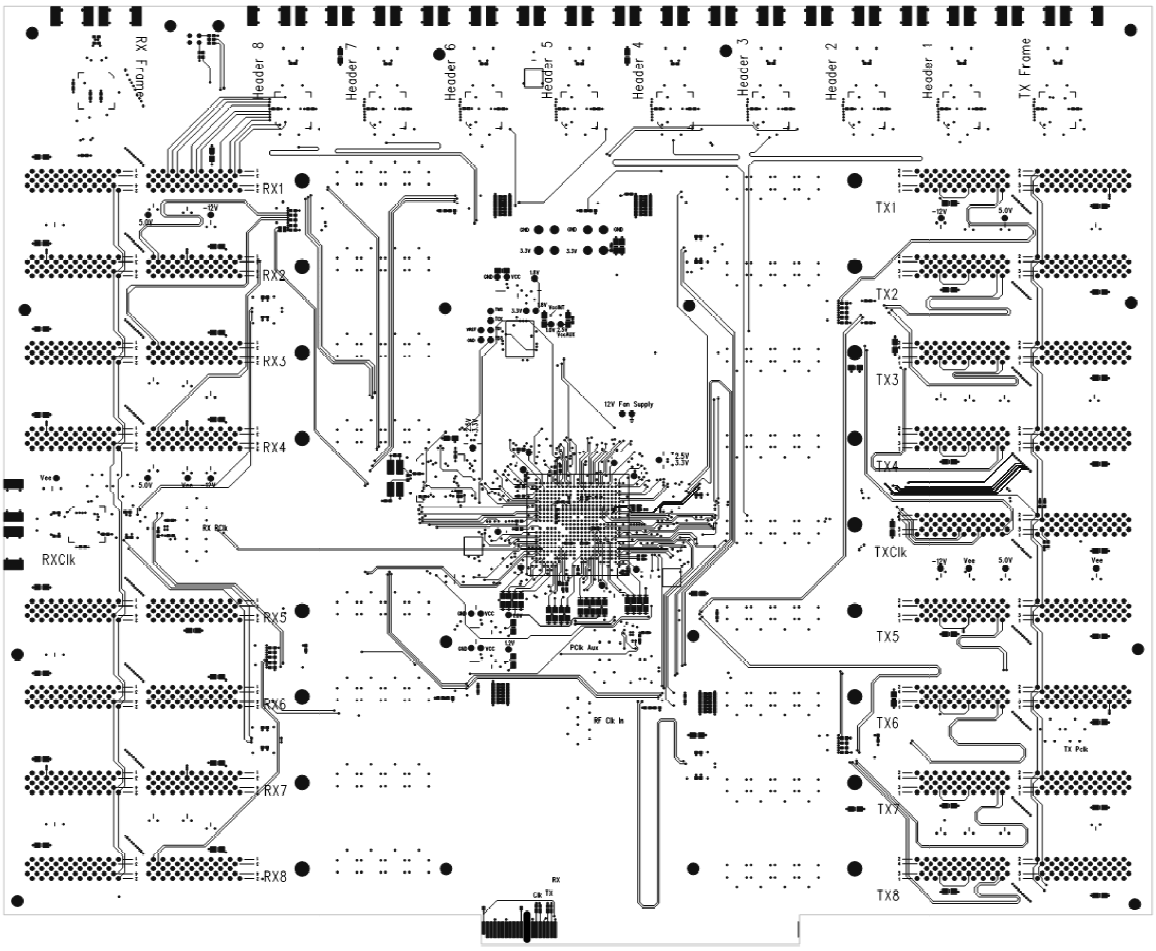
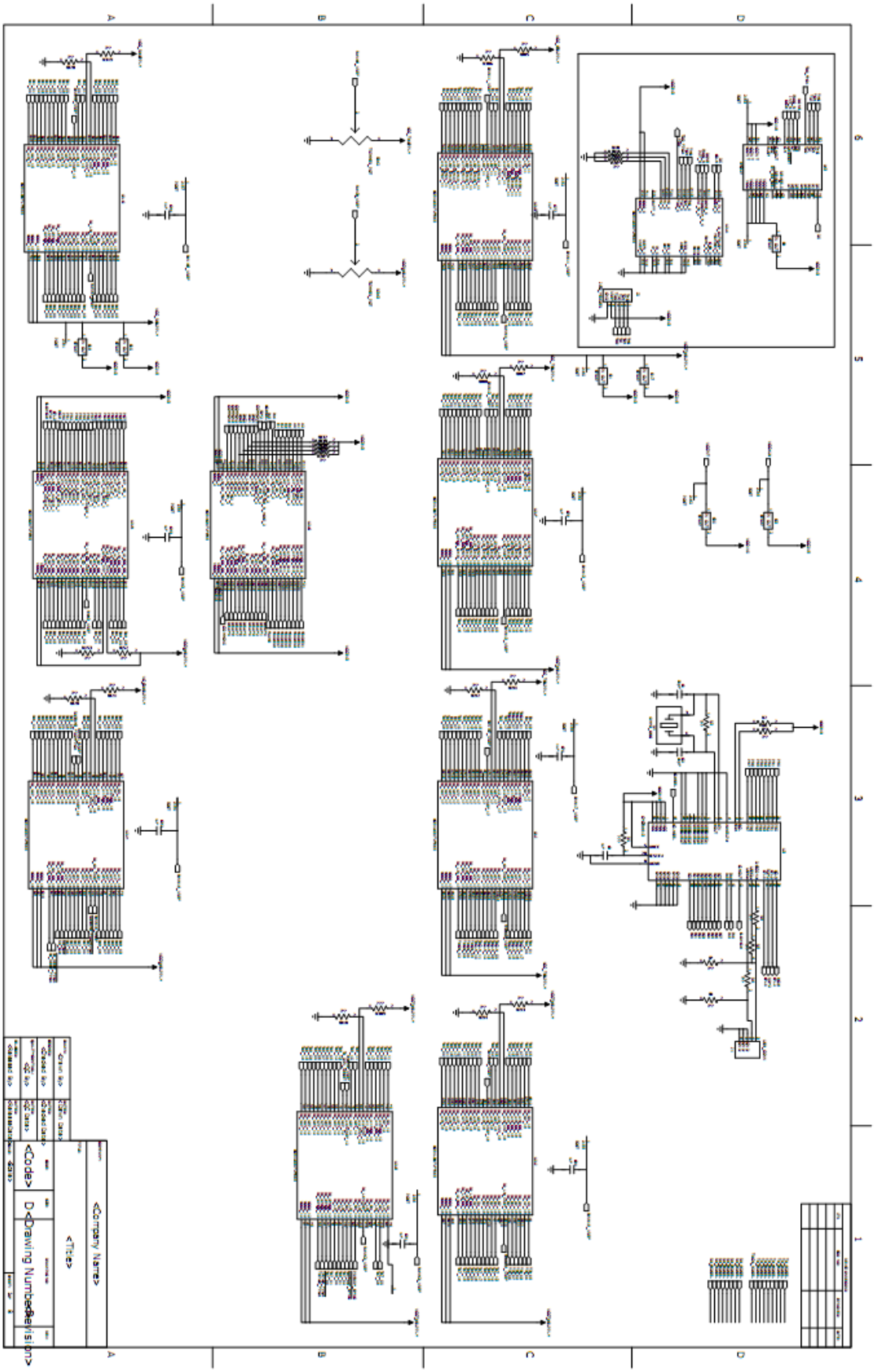
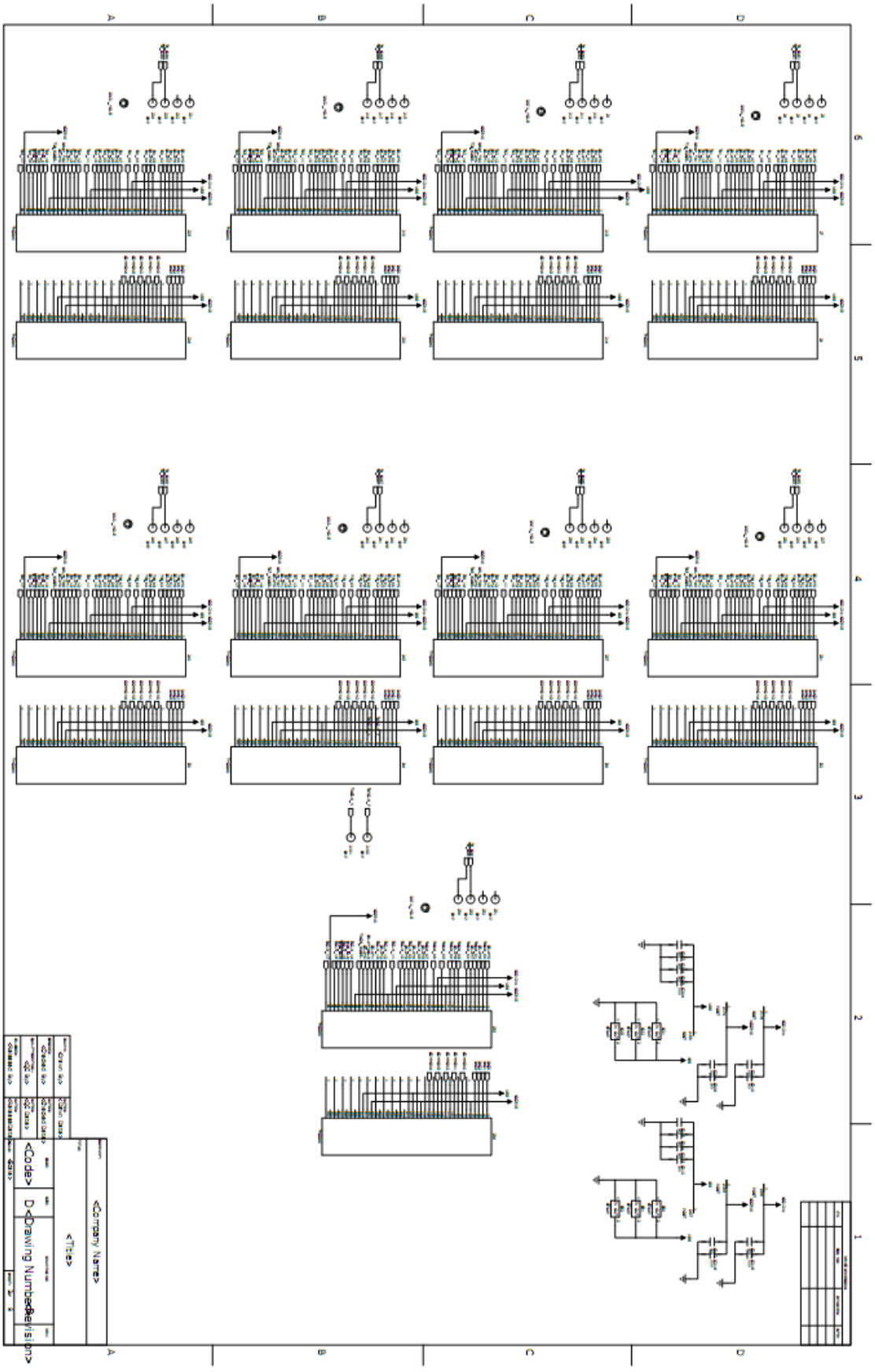


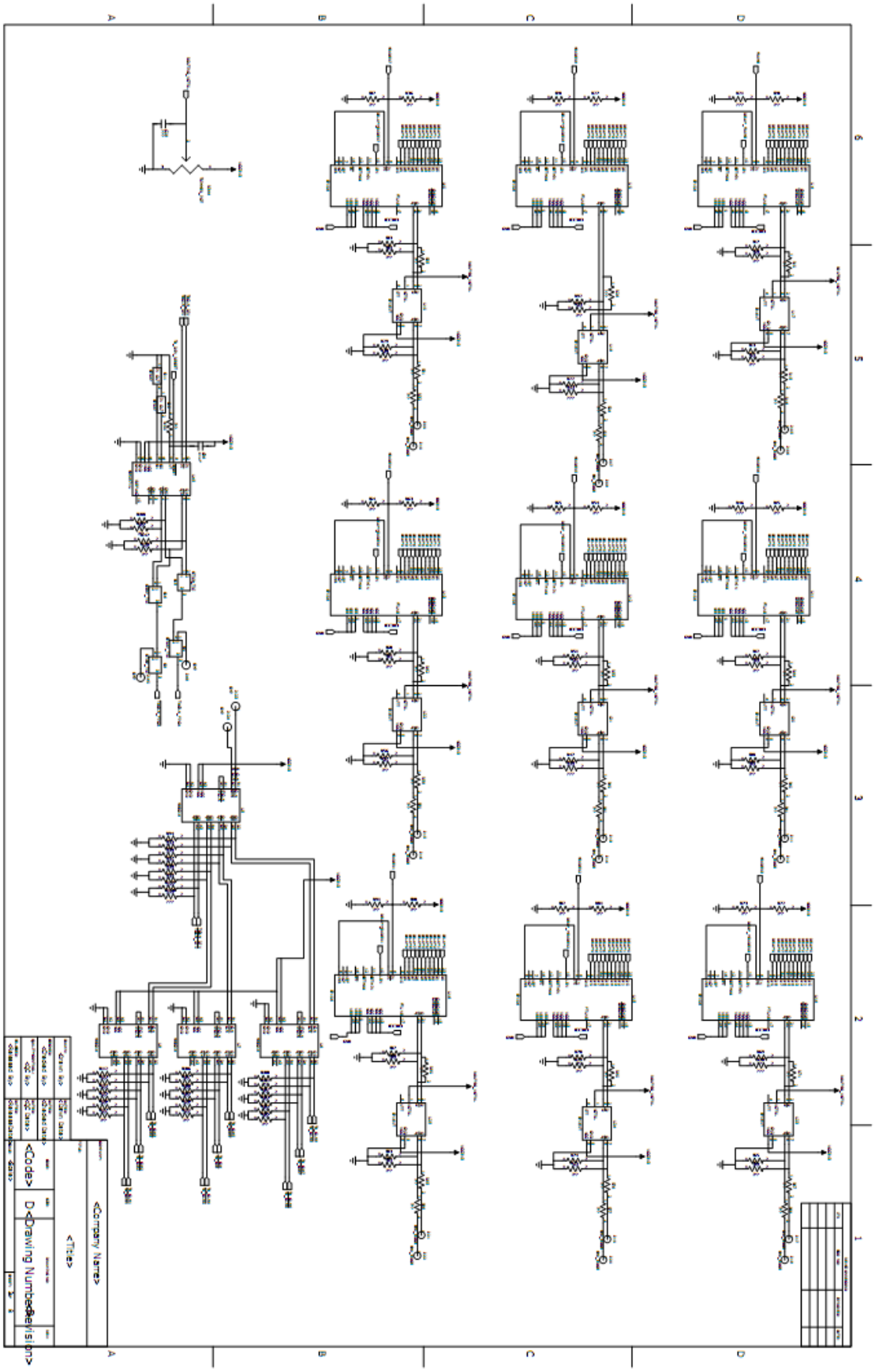
Figure A.7 Bottom layer

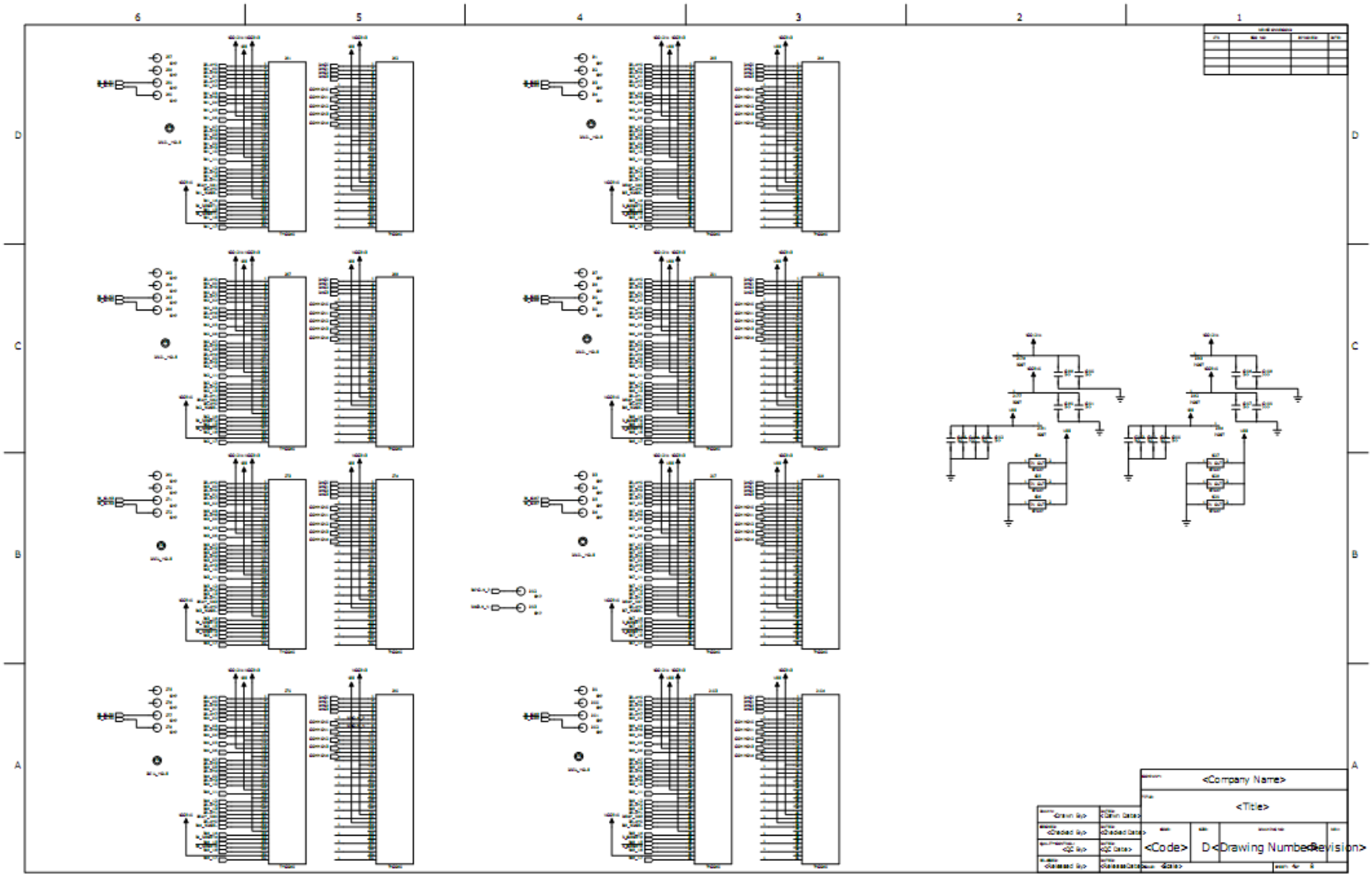
Schematic:

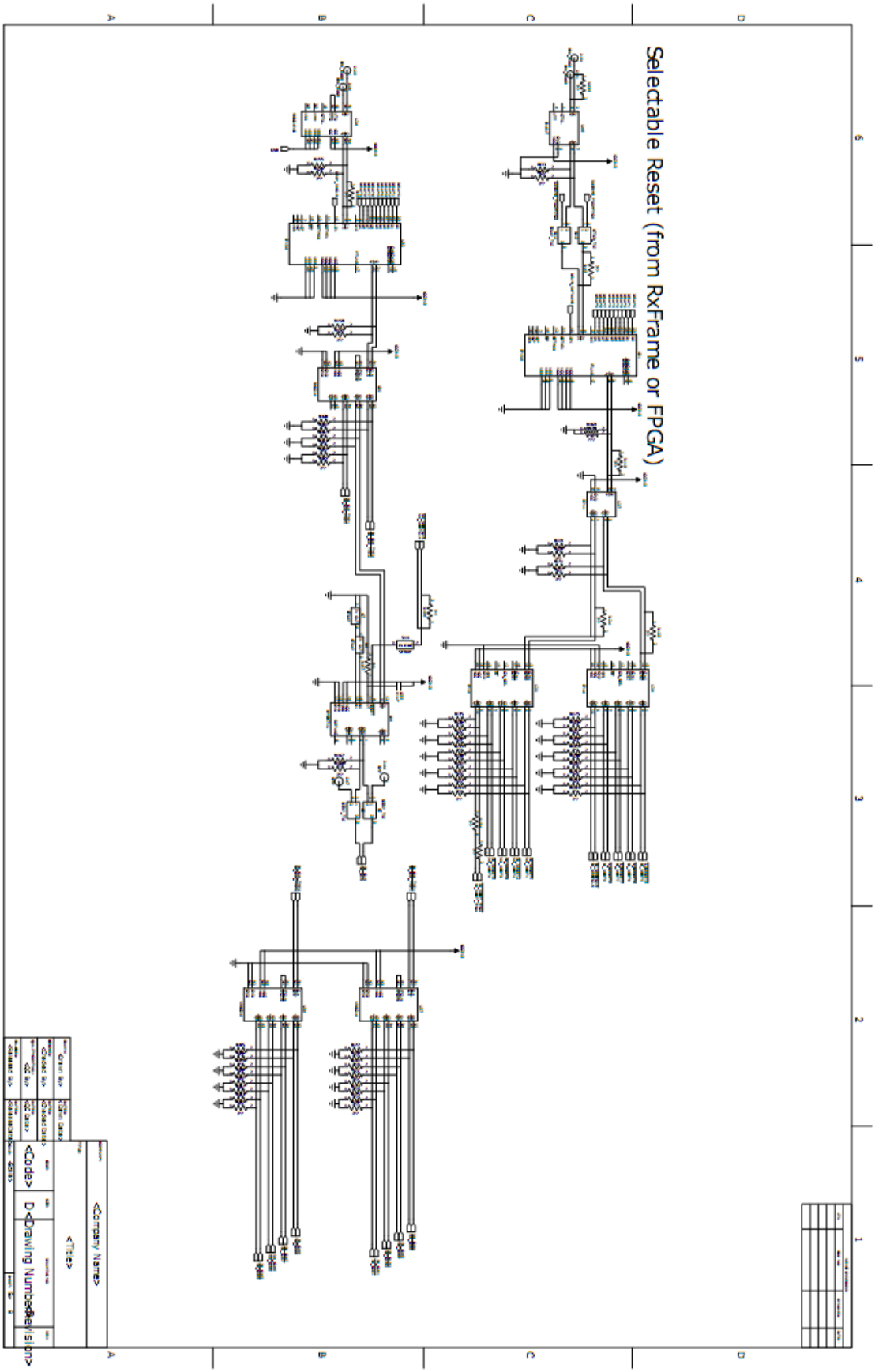
The following figures show the schematic level representation of the above system and how the various elements are interconnected at the device level.


<Title>
<Drawing Number>
<Size>
<Company Name>

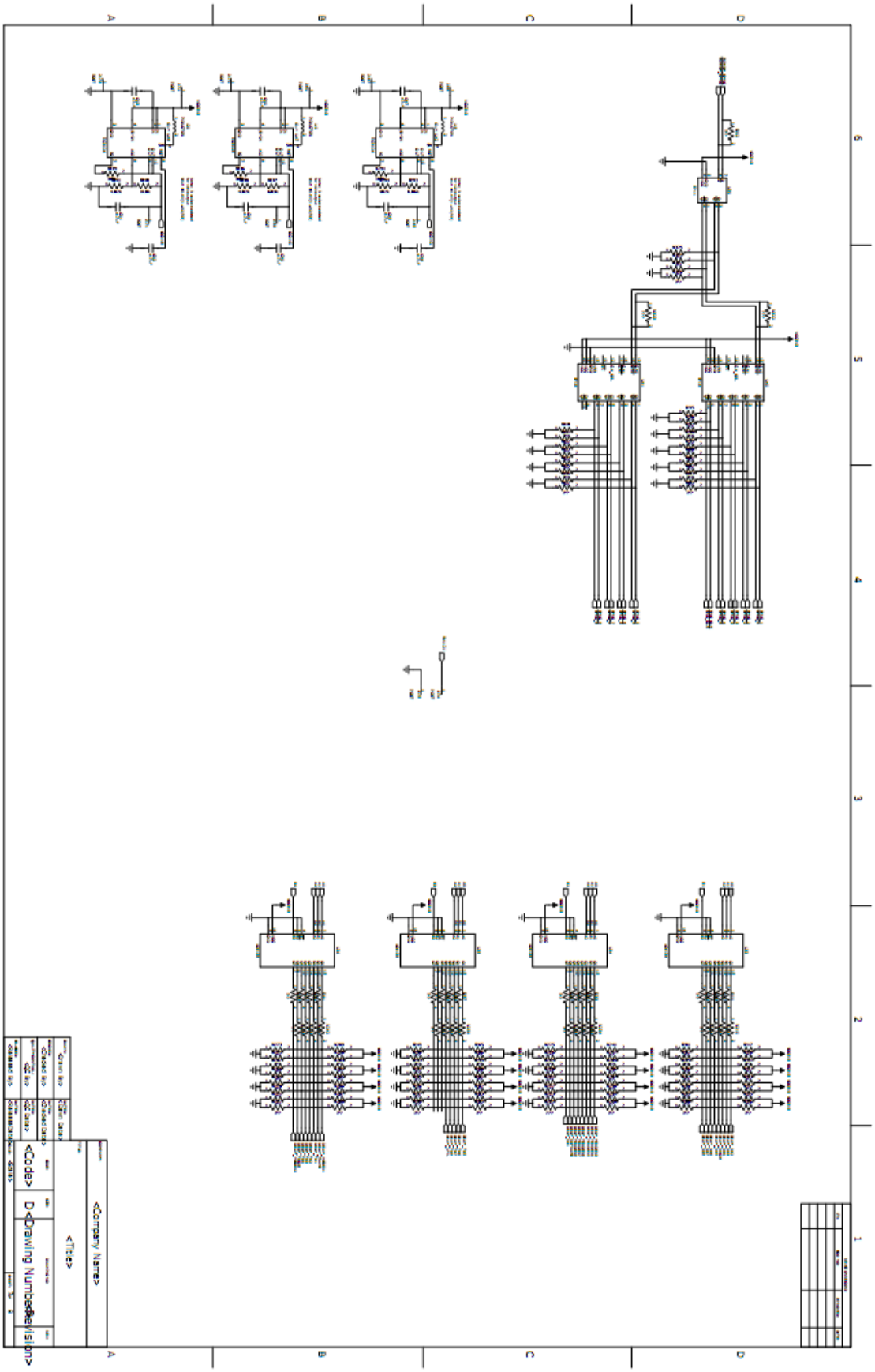

<Code>
<Drawing Number>
<Title>
<Company Name>

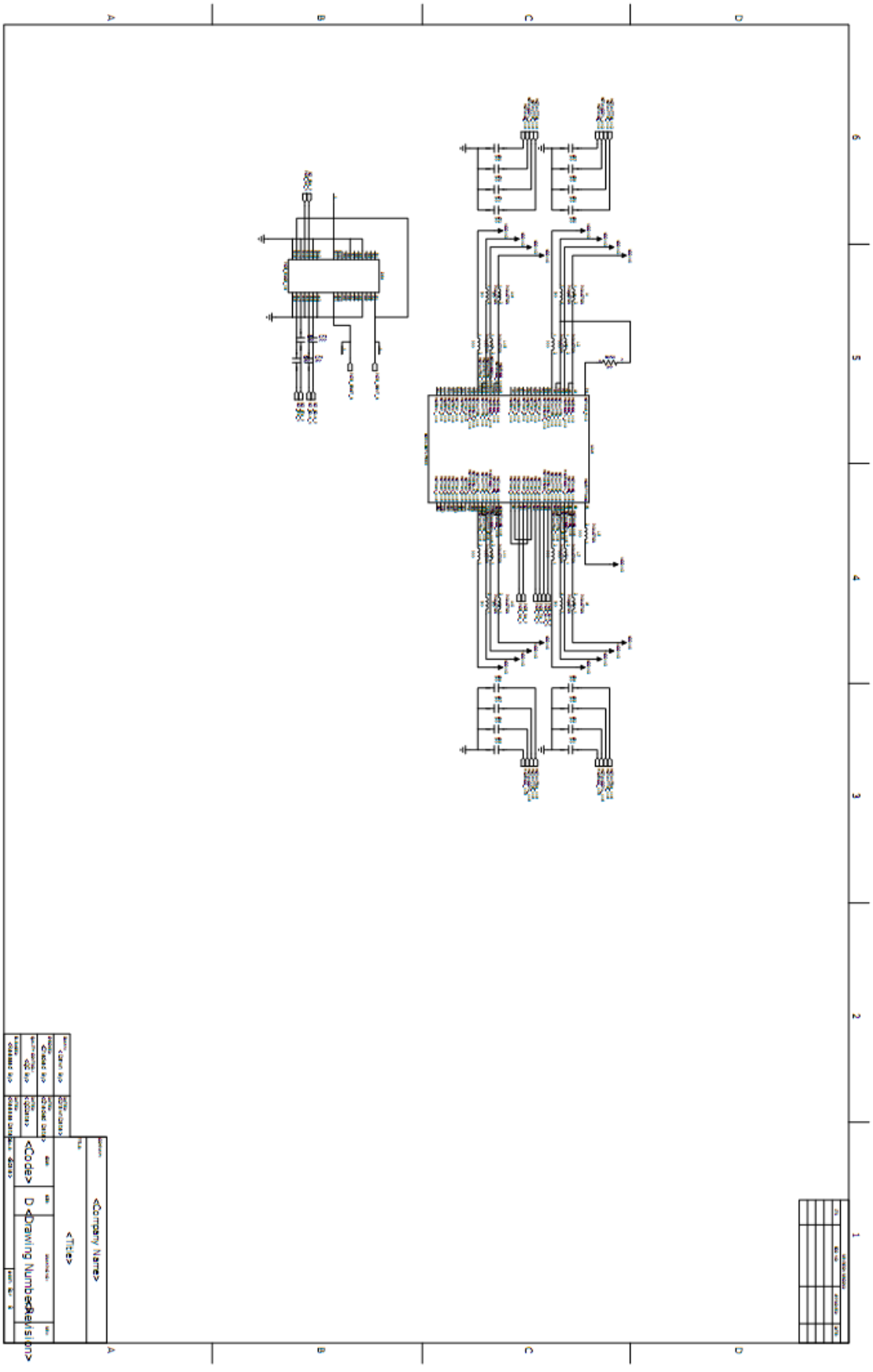

<Company Name>
<Title>
Drawing Number
<Code>


6
5
4
3
2
1
D
C
B
A
<Company Name>
<Title>
<Code>
D
<Drawing Number>
<Revision>


Selectable Reset (from RxFrame or FPGA)
<Company Name>
<Title>
Drawing Number
<Code>

<Company Name>
<Title>
Drawing Number
<Code>
<Revision>


<Company Name>
<Title>
D<Drawing Number>
<Code>
<Revision>


<Company Name>
<Title>
<Drawing Number>
<Code>
<Revision>

APPENDIX B

FPGA FIRMWARE

In this appendix, details of the FPGA programming logic used to connect to external interfaces, generate and process high-speed data to and from the extension modules, and to control the overall system is presented. The FPGA firmware is developed in the Xilinx ISE software suite. A mixture of schematic level design entry and HDL design modules can be utilized to produce the final result. As the FPGA design is variable to adapt to the particular extension modules populated in the test platform, the design presented here corresponds to a full complement of 2.5 Gbps TX and RX modules intended to represent a single injection and egress port of the Data Vortex as accessed over the USB interface.

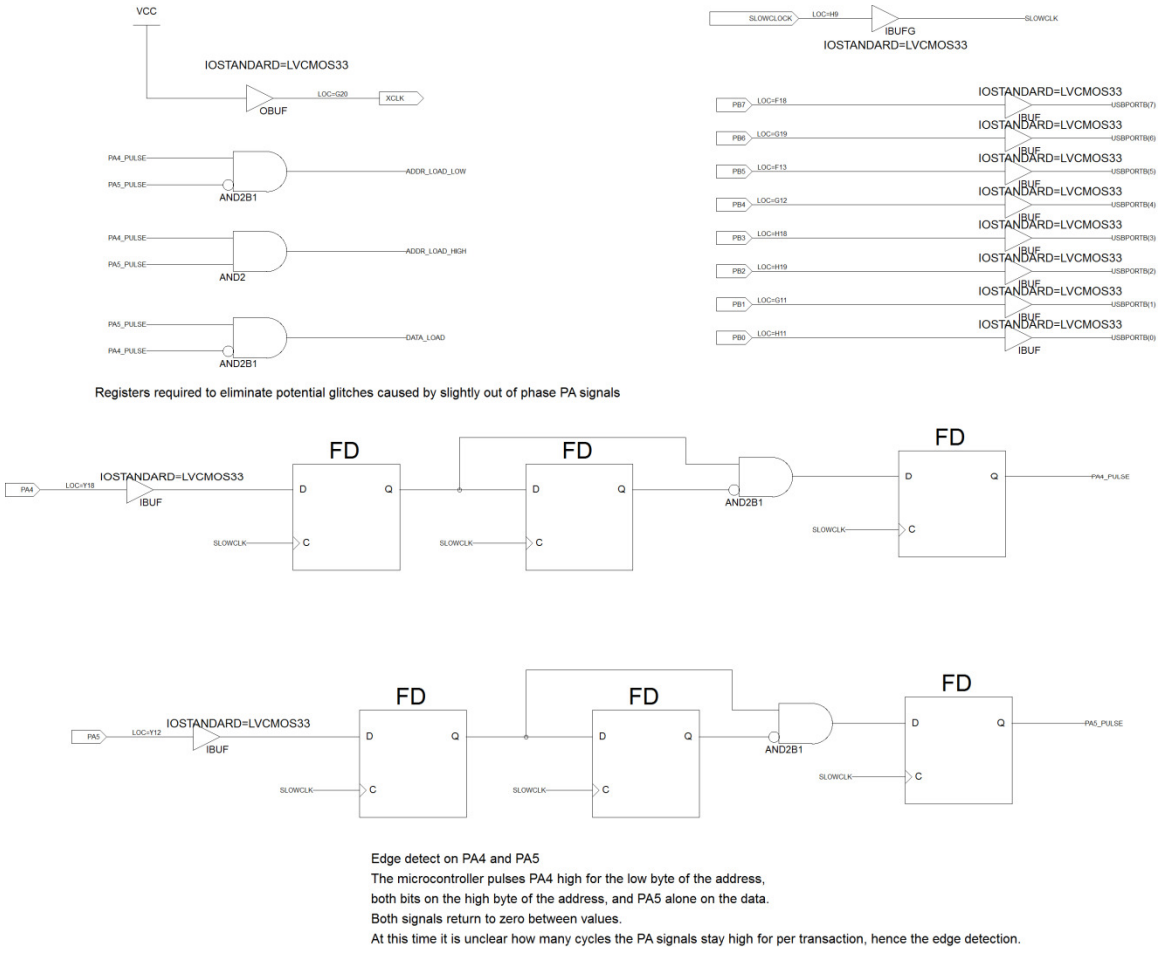


Figure B.1 USB interface, part 1.

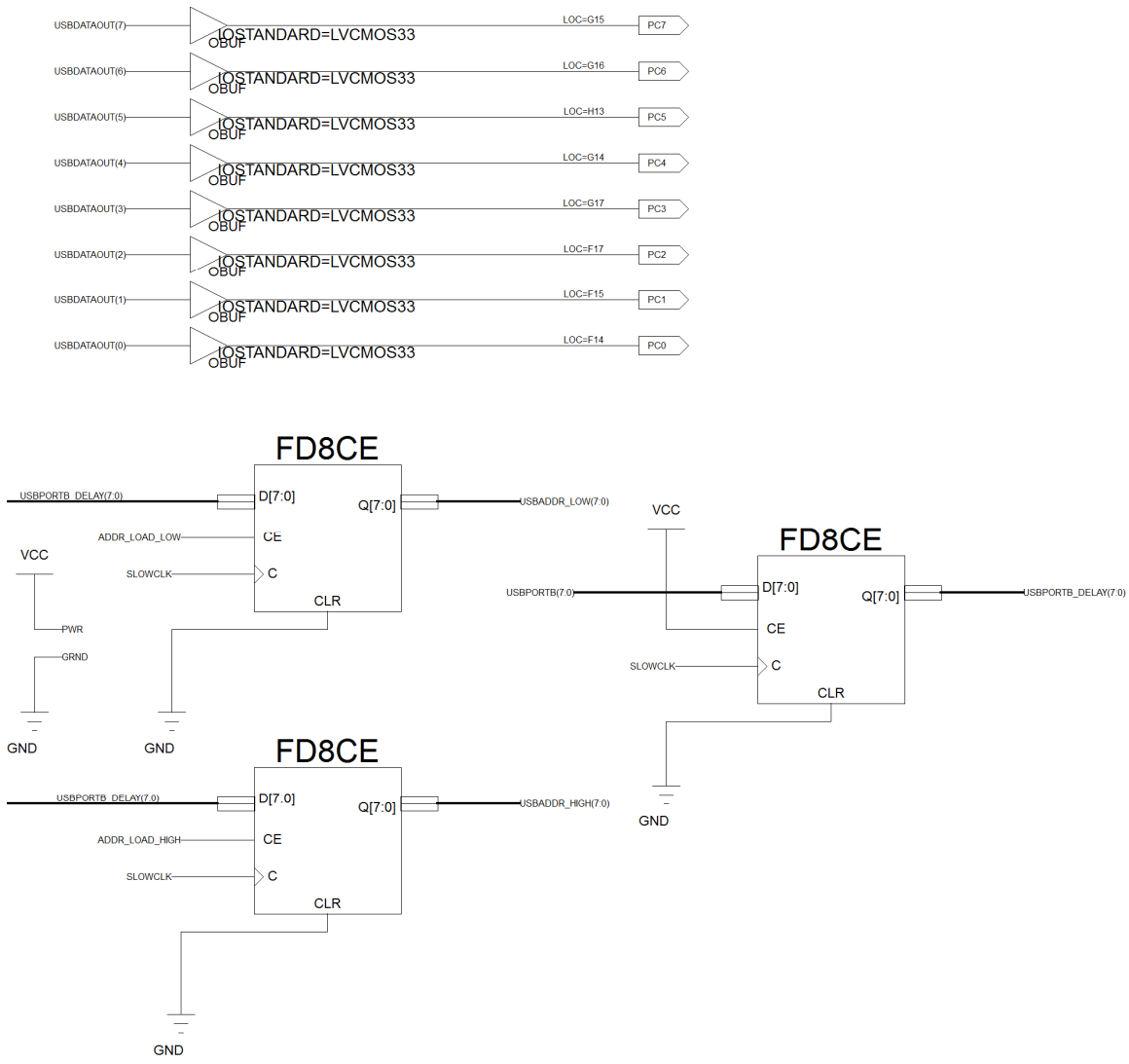


Figure B.2 USB interface, part 2.

These two figures represent the individual data pins between the FPGA and the Cypress microcontroller and corresponding logic required to interface to said device. The interface is implemented as a pair of address bytes, a data byte, and DATA_LOAD which functions as a write enable on incoming data writes. Read operations are passively initiated by pushing an address to the FPGA which subsequently presents the data on the outgoing pins to be sampled at a later time by the microcontroller.

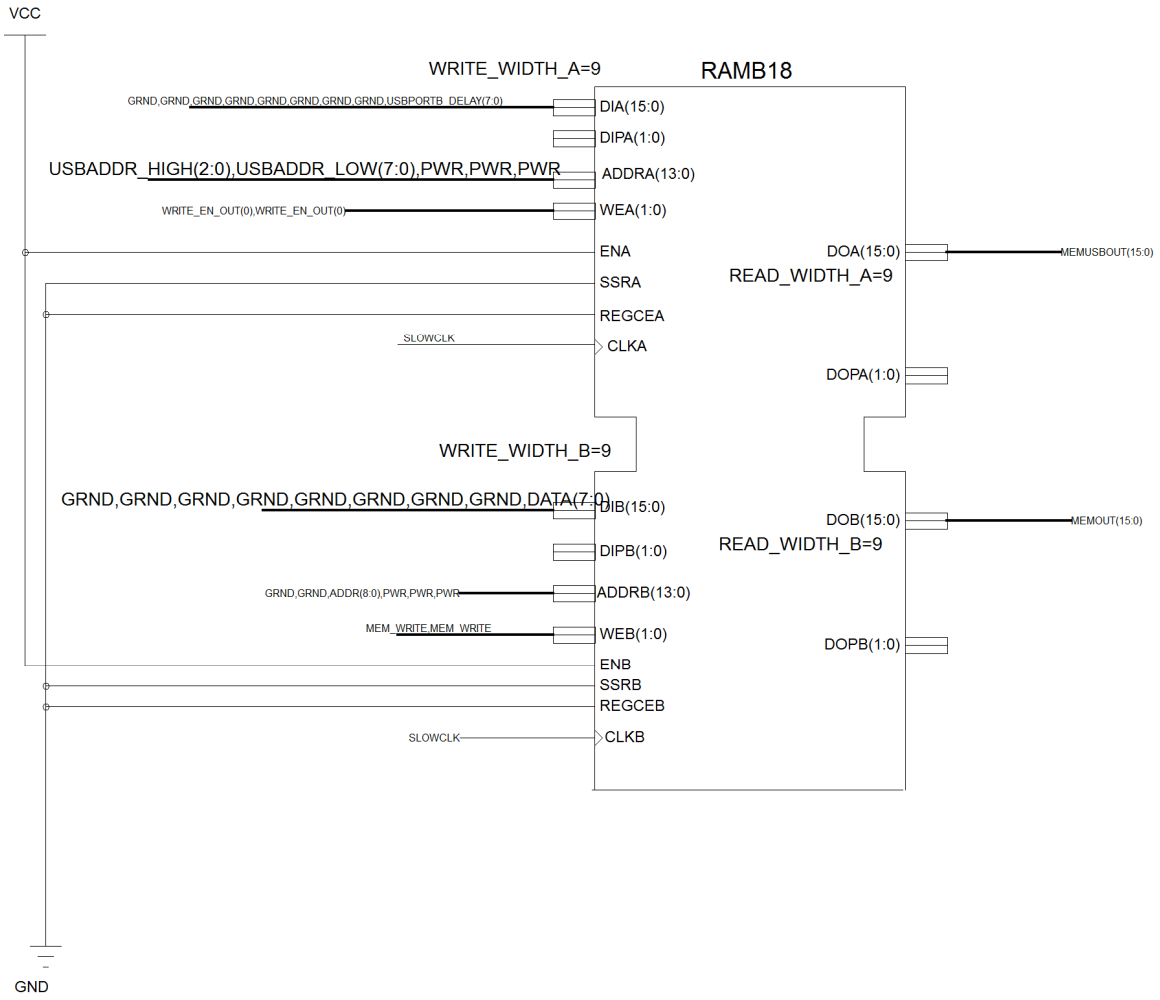


Figure B.3 Control, configuration, and command memory.

This figure shows one of the many supported memory blocks within the design. This particular memory block is referred to as Block 0. The upper port, Port A, is tied to the global USB address and data system. Writes occur to this memory only when WRITE_EN_OUT(0) is high, as decoded by the logic block in the next figure. Port B is operated in conjunction with the controller logic, processing instructions or applying configuration values such as timing delays which are stored in this memory.

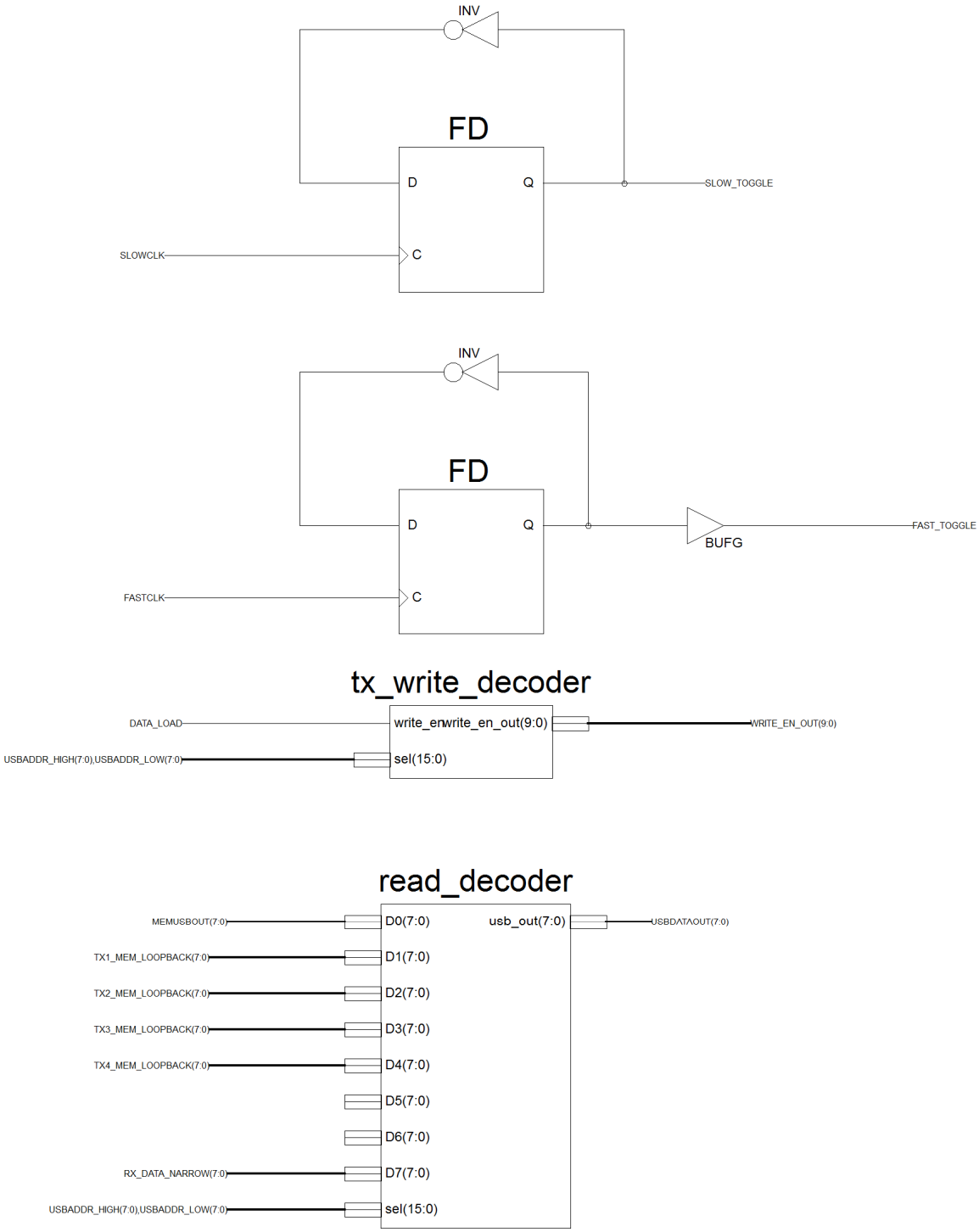


Figure B.4 Write and read address decoders.

There are 10 primary writable memory blocks in this design implementation. TX_WRITE_DECODER processes the incoming address and DATA_LOAD signal to

activate the appropriate enable bits for the desired memory destinations. This decoder can be expanded as required to support additional memory destinations. A read decoder is similarly used to allow the reading of the various memories or data storage elements in the system.

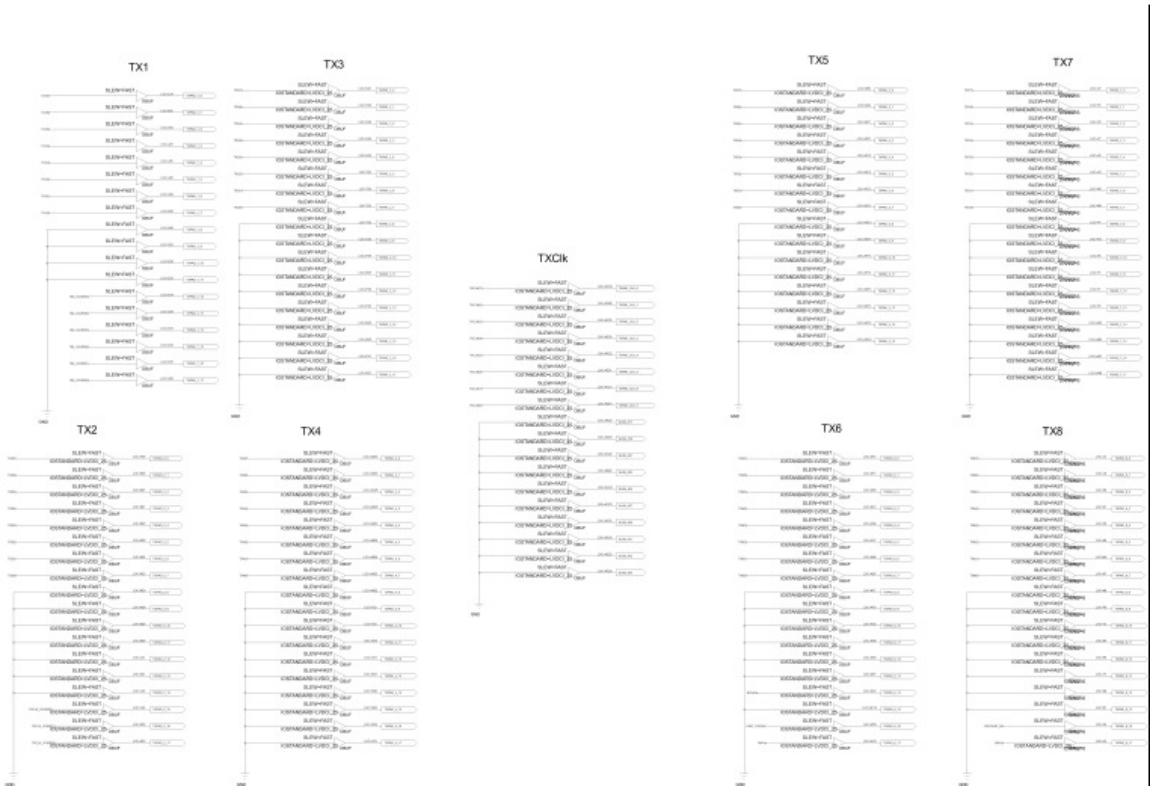


Figure B.5 Bank A data pin configuration.

The figure above shows the general purpose I/O pin definitions for the Bank A module slots. A zoom in for the center block, utilized in this design as TxClk, is shown below. For this particular design, the standard 2.5 Gbps single-channel modules are utilized which only require 8 of the total 18 data pins. Unused pins are tied to ground.

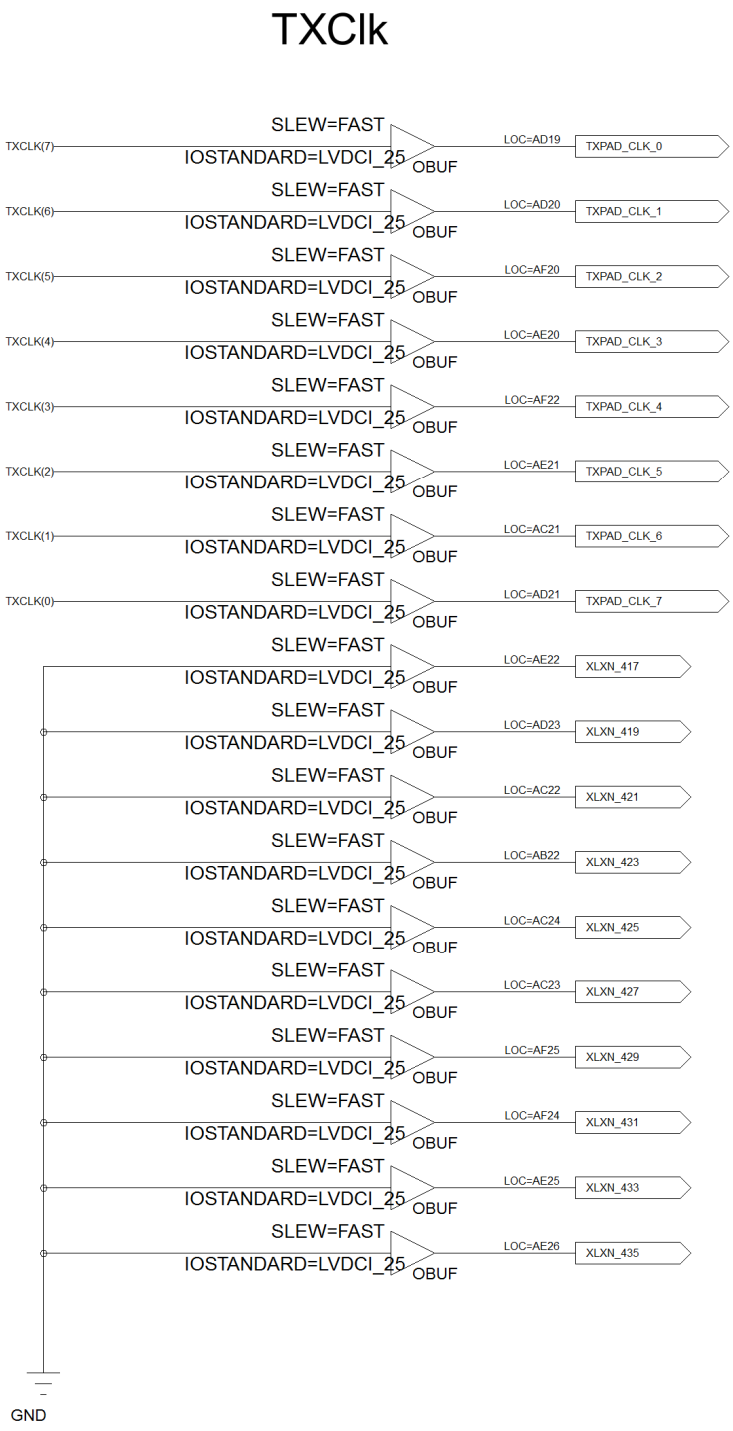


Figure B.6 Single Bank A data pin configuration.

Zoom-in of the pins for the TxClk channel. LVDCI_25 is a 2.5V based, low-voltage and source controlled impedance signaling configuration supported by the FPGA

which is relatively compatible with the moderate-speed signals between the FPGA and PECL logic on the modules.

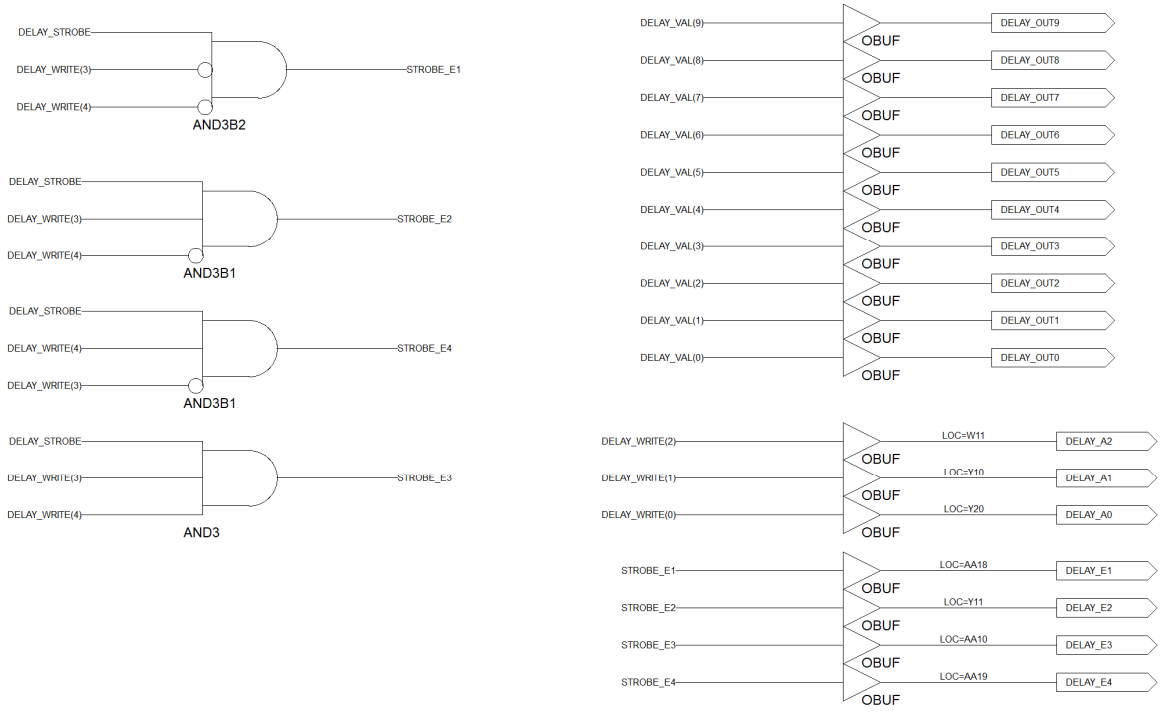


Figure B.7 Delay control data and enable pins.

Delay programming pins. 3 to 8 decoder chips are on the main board.

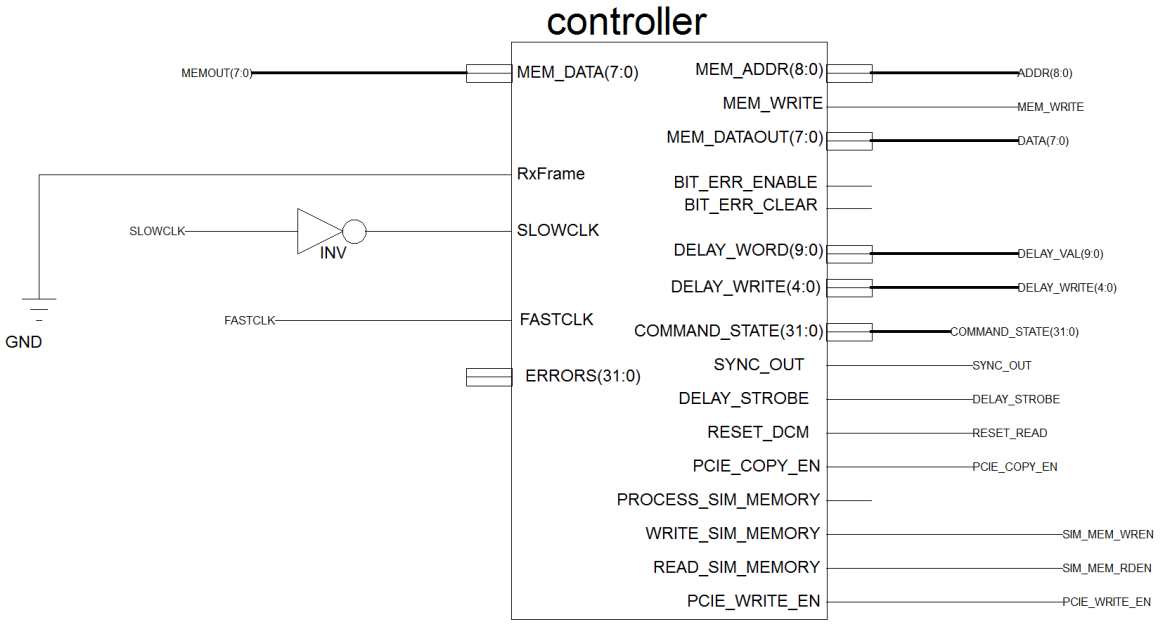


Figure B.8 Primary system controller.

The controller block manages the Block 0 memory, moving configuration values into appropriate globally viewable registers (COMMAND_STATE(31:0)), stores delay values, and processes commands as initiated across the external interface. Some operations, such as delay writing, are directly controlled by this module as shown by the DELAY_WORD and DELAY_WRITE signals, while other operations are simply initiated by this module and carried through to completion by additional modules.

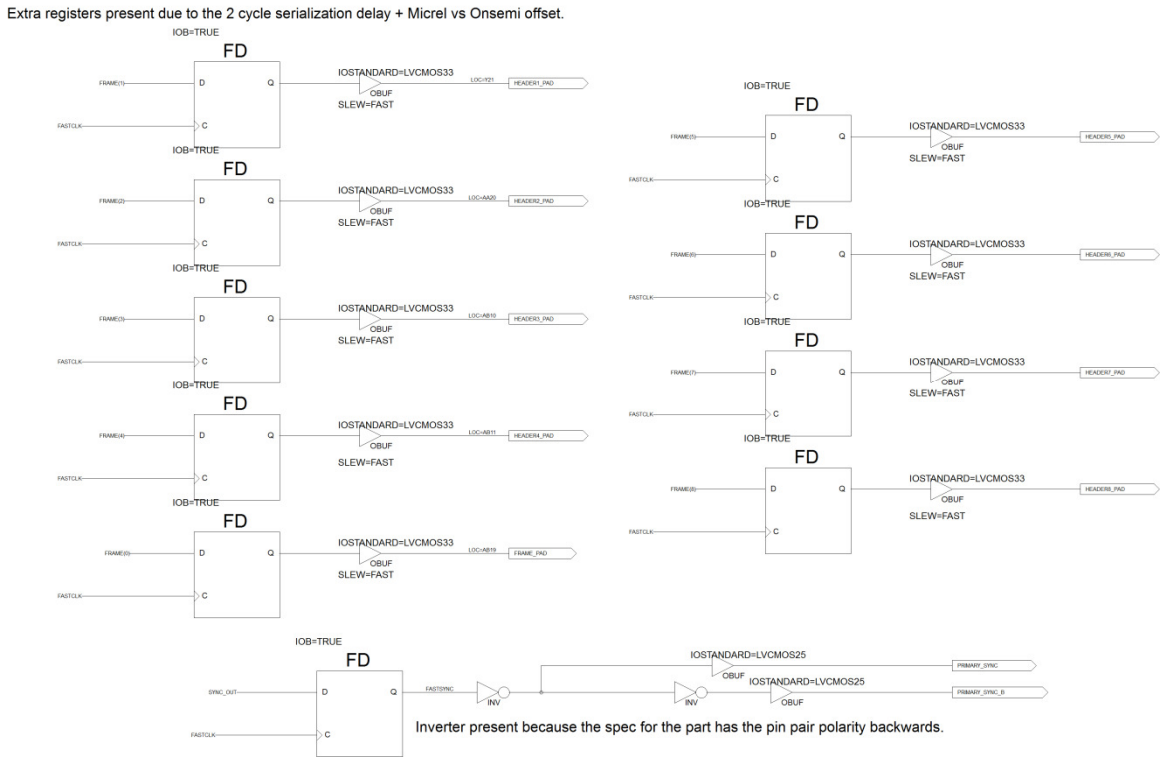


Figure B.9 Frame, routing, and synchronization pin elements.

The frame and routing bits are handled externally as single-ended signals, while the synchronization signal is distributed on the test platform as a differential signal. These signals are registered in storage elements physically incorporated into the FPGA I/O clusters to minimize the amount of skew experienced between the signals at the board level.

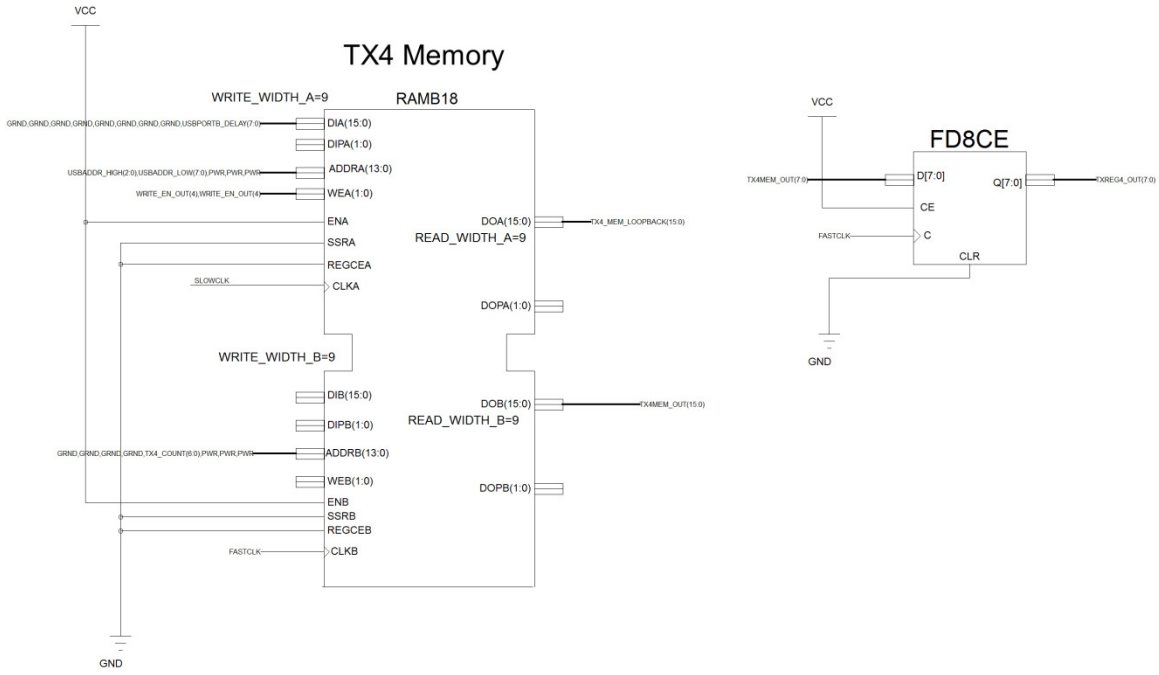


Figure B.10 Stored pattern memory for a single transmitter (Bank A) channel.

A zoom-in of one of the memory blocks used for storage of the transmitted bit patterns. Data is stored as a sequential series of bytes throughout the memory. These values can be read and written across the USB interface through the A port while full speed application logic accesses the B port for reading only. This port is passed through a secondary pipeline register to ensure that the values can transition across the FPGA at the desired clock rate. If timing constraints cannot be met for one or more collection of signals, additional register stages can be added in an attempt to improve system timing.

Additional memories are similarly implemented for each of the remaining transmitter data channels.

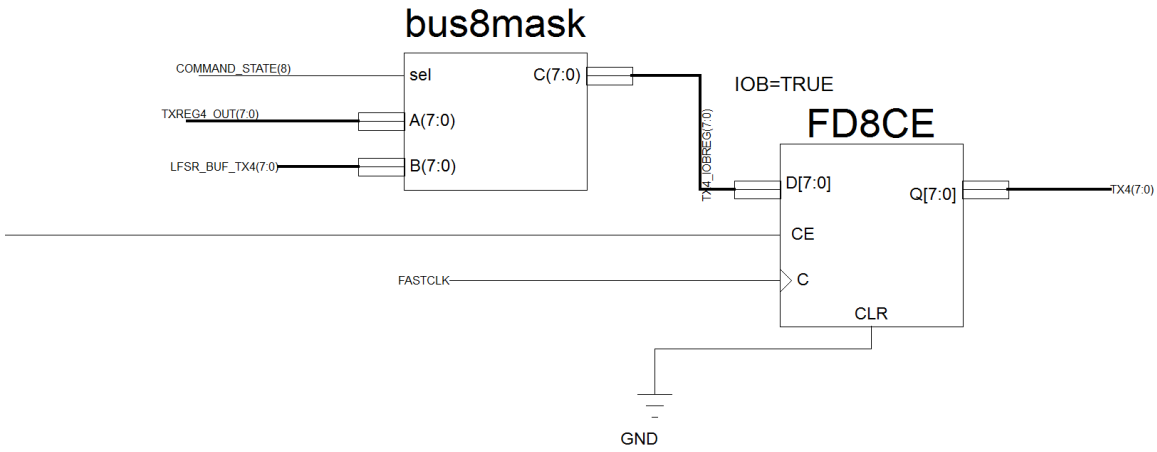


Figure B.11 Psuedo-random data injection.

This configuration, specific to the TX4 channel but similarly implemented as needed for the other transmitter channels, shows how pseudo-random data can be injected into the signal path. The bus8mask module allows for the output of the memory module to be passed unchanged or combined in a logical AND operation with pseudo-random data generated from an LFSR (not shown). Where 1's appear in the stored memory pattern, the LFSR data will be propagated forward through the signal path, allowing the stored memory patterns to be interpreted as a data mask. This allows for the creation of arbitrarily shaped bursts of randomized data. In the Data Vortex testing applications, this may be the centralized 32 bits of packet data or is conformable to any memory sequence that can be programmed.

Alternatively, a bus8mux can be utilized to completely switch between the stored patterns and the pseudo-random data if unshaped, continuous run sequences are desired. Both modules are dynamic and can be turned on or off with a single control bit which corresponds to COMMAND_STATE(8) in this case. This bit can be written or cleared as needed from the control interface or replaced with logic controlled by the FPGA.

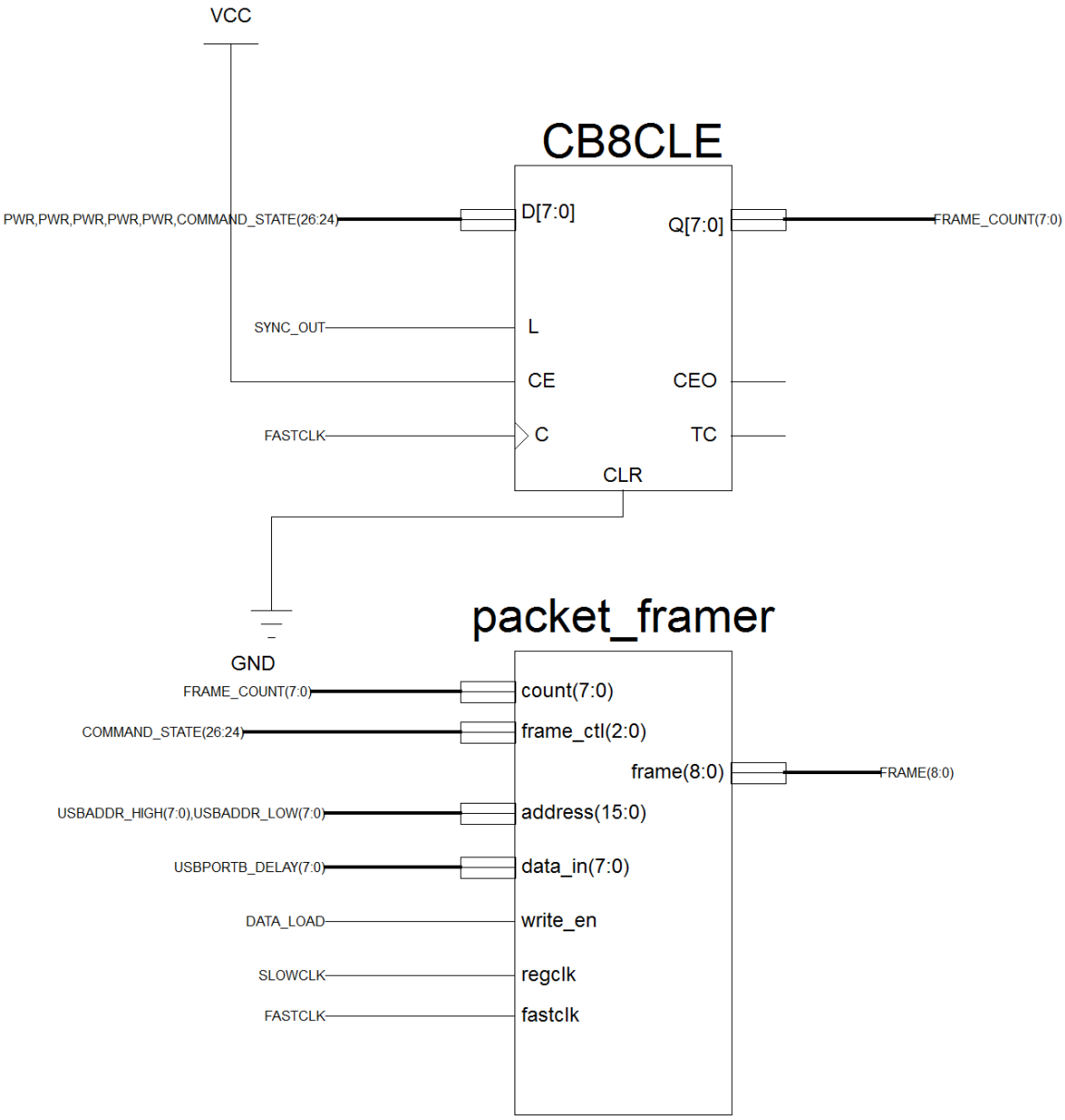


Figure B.12 Packetization logic.

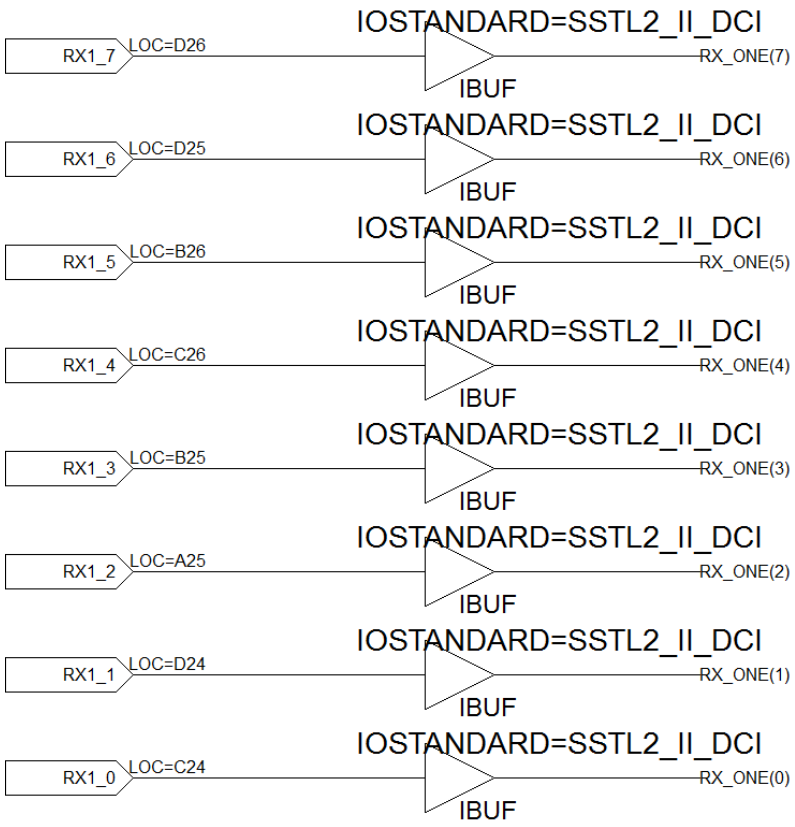


Figure B.13 Single channel receiver (Bank B) data pins.

A zoom-in of 8 of the 10 (unused pins are not shown here) general purpose I/O signals available on a single Bank B module, in this case designated as RX1 or the first receiver data channel. These pins are used in this design to capture the input response from the deserialization modules. The rx_buffer_shift module shown in the next figure, implemented one per Bank B module, takes the respective 8 bits and pushes them into a 64 element deep byte-wise shift register. The counter, CB4CE, increments on each incoming frame signal allowing for a total of 8 collected packets to be read and processed later.

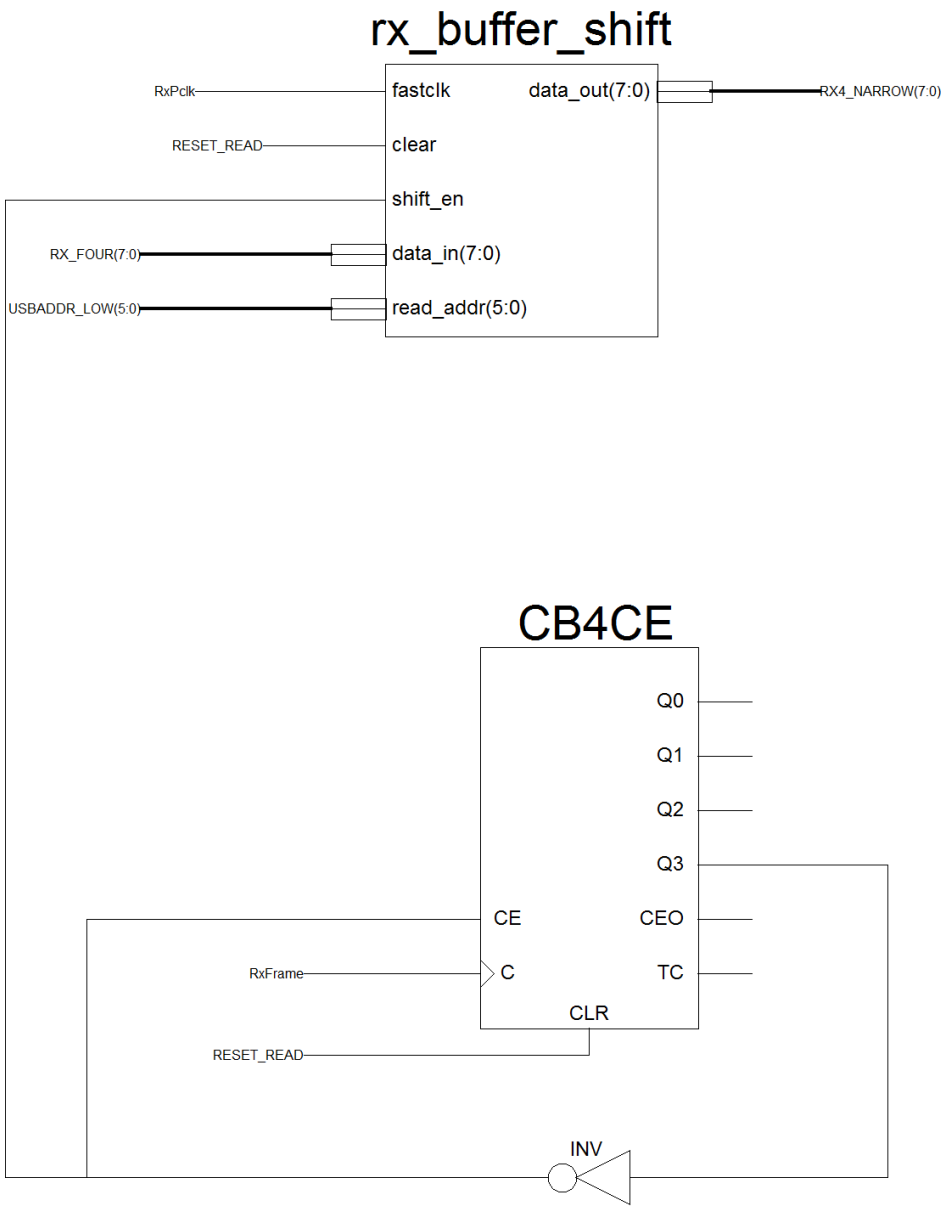


Figure B.14 Single channel receiver data buffer and packet counter.

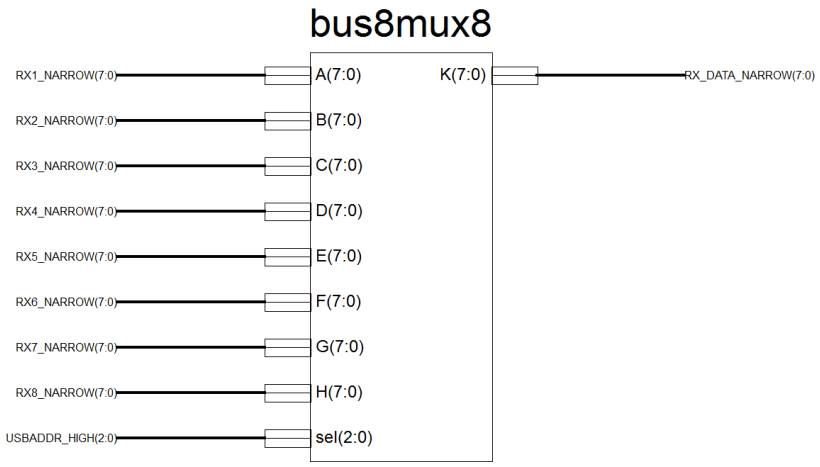


Figure B.15 RX buffer read decoder.

The bus8mux8 element shown here allows for the byte-wise output of the capture shift registers to be read through the data interface. The RX channel to be active is selected using the appropriate low bits in the high address byte. The output signal, RX_DATA_NARROW, is an input option to the read decoder previously shown in Figure B.4. Through a system of these decoders and an accurate map of the memory topology, any data values in the system can be made available to the external interface or function logic at any other point within the FPGA.

This section includes some of the critical components in VHDL form. The Xilinx ISE design environment allows for schematic representations, shown above, to be used interchangeably with HDL modules such as the VHDL shown here or similar implementations in Verilog depending on the author's preference.

Controller.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

--  Uncomment the following lines to use the declarations that are
--  provided for instantiating Xilinx primitive components.
--library UNISIM;
--use UNISIM.VComponents.all;

entity controller is
    Port ( MEM_DATA : in std_logic_vector(7 downto 0);
           MEM_ADDR : out std_logic_vector(8 downto 0);
           MEM_WRITE : out std_logic;
           SLOWCLK : in std_logic;
           MEM_DATAOUT : out std_logic_vector(7 downto 0);
        DELAY_WORD : out std_logic_vector(9 downto 0);
        DELAY_WRITE : out std_logic_vector(4 downto 0);
        COMMAND_STATE : out std_logic_vector(31 downto 0);
        DELAY_STROBE : out std_logic;
        FASTCLK : in std_logic;
        SYNC_OUT : out std_logic;
        ERRORS : in std_logic_vector(31 downto 0);
        RxFrame : in std_logic;
        BIT_ERR_CLEAR : out std_logic;
        BIT_ERR_ENABLE : out std_logic;
        RESET_DCM : out std_logic;
        PCIE_COPY_EN : out std_logic;
        PROCESS_SIM_MEMORY : out std_logic;
        WRITE_SIM_MEMORY : out std_logic;
        READ_SIM_MEMORY : out std_logic;
        PCIE_WRITE_EN : out std_logic);
end controller;

architecture Behavioral of controller is


type state_type is (RESET, STANDBY, DELAY_LOAD,SYNCHRONIZE,SYNCHRONIZE_DCM,PCIE_COPY_SET,
    PCIE_COPY_CLR,COMMAND_CLR,SIM_MEM_PROCESS,SIM_MEM_WRITE,SIM_MEM_READ,PCIE_WRITE_SET,
    PCIE_WRITE_CLR);
signal state : state_type;

signal delay_count : std_logic_vector (4 downto 0);
signal low_word : std_logic;
signal sub_state : std_logic_vector(2 downto 0);

signal sync : std_logic;
signal sync_dcm : std_logic;

signal temp_memaddr : std_logic_vector(7 downto 0);
signal delay_strobe_out : std_logic;

signal global_reset : std_logic;

signal toggle : std_logic;
```

```
begin

MEM_ADDR <= "0" & temp_memaddr;
delay_strobe <= delay_strobe_out;

BIT_ERR_CLEAR <= '0';
BIT_ERR_ENABLE <= '0';

sync_out <= sync;
reset_dcm <= sync_dcm;

--main state update
process(SLOWCLK,MEM_DATA,global_reset)
begin
    if(global_reset = '1') then
      global_reset <= '0';
    elsif(rising_edge(SLOWCLK)) then
      case state is
        when STANDBY =>   -- state STANDBY is the default "have nothing to do,
                    -- waiting for instruction" state
          MEM_WRITE <= '0';
          -- while in the STANDBY state, every time through the loop copy
          -- the values in memory 2,3,4, and 5 into a static register set visible
          -- outside of this module.  These bits are used to control
          case sub_state is
            when "000" =>
              temp_memaddr <= "00000010";
              sub_state <= "001";
            when "001" =>
              COMMAND_STATE(7 downto 0) <= MEM_DATA;
              temp_memaddr <= "00000011";
              sub_state <= "010";
            when "010"=>
              COMMAND_STATE(15 downto 8) <= MEM_DATA;
              temp_memaddr <= "00000100";
              sub_state <= "011";
            when "011"=>
              COMMAND_STATE(23 downto 16) <= MEM_DATA;
              temp_memaddr <= "00000101";
              sub_state <= "100";
            when "100"=>
              COMMAND_STATE(31 downto 24) <= MEM_DATA;
              sub_state <= "101";
-- COMMAND_STATE register bits are intended for persistent system affecting data,
-- select this or activate that a better and more dynamic solution for the future
-- would be to have a resizeable vector and a specific instruction to set/clear
-- bits in that vector.  'push' the address to modify to memory, execute a
-- set/clear/toggle instruction added to the instruction set below
            when "101" =>
              temp_memaddr <= "00000000";
              sub_state <= "110";
            when "110" =>
              if(MEM_DATA = "00000010") then
                state <= DELAY_LOAD;  -- instruction 2:  Delay load
                delay_count <= "00000";  -- track the # of delays loaded
                temp_memaddr <= "00000110";  -- memory address of first data => 6
                delay_strobe_out <= '0';  -- strobe the load bit on high bytes only
                low_word <= '0';  -- start on a low byte
              elsif(MEM_DATA = "00000011") then
                 state  <= SYNCHRONIZE;  -- instruction 3:  Synchronize
              elsif(MEM_DATA = "00000100") then
                 state  <= SYNCHRONIZE_DCM;  -- instruction 4:  Reset the DCM
              elsif(MEM_DATA = "00000101") then
                 state  <= PCIE_COPY_SET;   -- instruction 5-11:  PCIE demo instructions
              elsif(MEM_DATA = "00000110") then
                 state  <= PCIE_COPY_CLR;
              elsif(MEM_DATA = "00000111") then
                 state  <= SIM_MEM_PROCESS;
              elsif(MEM_DATA = "00001000") then
```

```
       state  <= SIM_MEM_WRITE;
    elsif(MEM_DATA = "00001001") then
       state  <= SIM_MEM_READ;
    elsif(MEM_DATA = "00001010") then
       state  <= PCIE_WRITE_SET;
    elsif(MEM_DATA = "00001011") then
       state  <= PCIE_WRITE_CLR;
    end if;
    sub_state <= "000";
  when others =>
    sub_state <= "000";
  end case; -- case sub_sate
------------------
when DELAY_LOAD =>
  if(delay_count = "11110") then  -- 30 total delays to program
    state <= COMMAND_CLR;  -- if done clear out and reset
  else
    if(low_word = '0') then  -- GET LOW BYTE, step 1
      delay_word(7 downto 0) <= mem_data(7 downto 0);  -- read the full byte
      low_word <= '1';  -- prep for the high byte
      delay_strobe_out <= '0';  -- don't pulse on low bytes
      temp_memaddr <= temp_memaddr + 1;  -- increment memory
      delay_write <= "00000";  -- delay target value
    elsif(delay_strobe_out = '1') then  -- CLEANUP, step 3
      delay_strobe_out <= '0';  -- done, clear LEN
      low_word <= '0';  -- prep for new low word
      delay_count <= delay_count + 1;  -- increment count
    else  -- GET HIGH BYTE, step 2
      delay_word(9 downto 8) <= mem_data(1 downto 0);  -- read the high byte
      temp_memaddr <= temp_memaddr + 1;  -- increment memory
      delay_strobe_out <= '1'; -- high byte, time to pulse
      delay_write <= delay_count;  -- delay target to write
    end if;
  end if;
------------------
when SYNCHRONIZE =>  -- set application specific execution bits high
    global_reset <= '1';
    sync <= '1';
    state <= COMMAND_CLR;  -- value will hold one cycle, clear on next
------------------
when SYNCHRONIZE_DCM =>
    sync_dcm <= '1';
    state <= COMMAND_CLR;
------------------
when PCIE_COPY_SET =>
    pcie_copy_en <= '1';
    state <= COMMAND_CLR;
------------------
when PCIE_COPY_CLR =>
    pcie_copy_en <= '0';
    state <= COMMAND_CLR;
------------------
when SIM_MEM_PROCESS =>
    process_sim_memory <= '1';
    state <= COMMAND_CLR;
------------------
when SIM_MEM_WRITE =>
    write_sim_memory <= '1';
    state <= COMMAND_CLR;
------------------
when SIM_MEM_READ =>
    read_sim_memory <= '1';
    state <= COMMAND_CLR;
------------------
when PCIE_WRITE_SET =>
    pcie_write_en <= '1';
    state <= COMMAND_CLR;
------------------
when PCIE_WRITE_CLR =>
    pcie_write_en <= '0';
    state <= COMMAND_CLR;
```

```
          -------------------
          when COMMAND_CLR =>  -- execution complete, clear application signals
              MEM_WRITE <= '1';
              sync <= '0';
              sync_dcm <= '0';
              MEM_DATAOUT <= "00000000";  -- mem value set to 0
              temp_memaddr <= "00000000";  -- mem address set to 0.  Instruction will be
cleared
              process_sim_memory <= '0';
              write_sim_memory <= '0';
              read_sim_memory <= '0';
              state <= STANDBY;
              sub_state <= "000";
          when others =>
            state <= STANDBY;
            sub_state <= "000";
          end case;
    end if;
end process;

end Behavioral;
```

bus8mux.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

--  Uncomment the following lines to use the declarations that are
--  provided for instantiating Xilinx primitive components.
--library UNISIM;
--use UNISIM.VComponents.all;

entity bus8mux is
    Port ( A : in std_logic_vector(7 downto 0);
           B : in std_logic_vector(7 downto 0);
           sel : in std_logic;
           C : out std_logic_vector(7 downto 0));
end bus8mux;

architecture Behavioral of bus8mux is

begin

process(sel,A,B)
begin
     IF( sel = '0' ) THEN
       C <= A;
       ELSE
       C <= B;
     END IF;
end process;

end Behavioral;
```

bus8mask.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

--  Uncomment the following lines to use the declarations that are
--  provided for instantiating Xilinx primitive components.
--library UNISIM;
```

```
--use UNISIM.VComponents.all;

entity bus8mask is
    Port ( A : in std_logic_vector(7 downto 0);
           B : in std_logic_vector(7 downto 0);
           sel : in std_logic;
           C : out std_logic_vector(7 downto 0));
end bus8mask;

architecture Behavioral of bus8mask is

begin

process(sel,A,B)
begin
     IF( sel = '0' ) THEN
       C <= A;
       ELSE
       C <= A and B;
     END IF;
end process;

end Behavioral;
```

lfsr_vhdl8.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

--  Uncomment the following lines to use the declarations that are
--  provided for instantiating Xilinx primitive components.
--library UNISIM;
--use UNISIM.VComponents.all;

entity lfsr_vhdl8 is
    Port ( reset : in std_logic;
           enable : in std_logic;
           clock : in std_logic;
        seed : in std_logic_vector(7 downto 0);
           pbsr : out std_logic_vector(7 downto 0)
       );
end lfsr_vhdl8;

architecture Behavioral of lfsr_vhdl8 is

signal lfsr : std_logic_vector(7 downto 0);

begin

pbsr <= lfsr;

process(clock,reset)
begin
  if(reset = '1') then
    lfsr <= seed;
  elsif(rising_edge(clock) AND (enable = '1')) then
    lfsr(7 downto 0) <= lfsr(6 downto 4) & (lfsr(7) XOR lfsr(3)) &
    (lfsr(7) XOR lfsr(2)) & (lfsr(7) XOR lfsr(1)) & (lfsr(0)) & (lfsr(7));
  end if;
end process;

end Behavioral;
```

read_decoder.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

--  Uncomment the following lines to use the declarations that are
--  provided for instantiating Xilinx primitive components.
--library UNISIM;
--use UNISIM.VComponents.all;

-- partially implemented read decoder, full 32:1 should be implemented
-- to ensure that all bank values are accessible
entity read_decoder is
    Port ( D0 : in std_logic_vector(7 downto 0);
           D1 : in std_logic_vector(7 downto 0);
        D2 : in std_logic_vector(7 downto 0);
        D3 : in std_logic_vector(7 downto 0);
        D4 : in std_logic_vector(7 downto 0);
        D5 : in std_logic_vector(7 downto 0);
        D6 : in std_logic_vector(7 downto 0);
        D7 : in std_logic_vector(7 downto 0);
           sel : in std_logic_vector(15 downto 0);
           usb_out : out std_logic_vector(7 downto 0));
end read_decoder;

architecture Behavioral of read_decoder is

begin

process(sel(14 downto 11),D0,D1,D2,D3,D4,D5,D6,D7)
begin
     case sel(14 downto 11) is
      when "0000"=>
        usb_out <= D0;
        when "0001" =>
        usb_out <= D1;
        when "0010" =>
        usb_out <= D2;
        when "0011" =>
        usb_out <= D3;
        when "0100" =>
        usb_out <= D4;
        when "0101" =>
        usb_out <= D5;
        when "0110" =>
        usb_out <= D6;
        when "1010" =>
        usb_out <= D7;
      when others =>
        usb_out <= "00000000";
      end case;
end process;

end Behavioral;
```

tx_register_decode.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

--  Uncomment the following lines to use the declarations that are
--  provided for instantiating Xilinx primitive components.
--library UNISIM;
--use UNISIM.VComponents.all;

-- This module decodes the respective narrow (bytewise) and wide (64bit pcie) addresses
-- This module is not active or needed if only the bytewise memory system is utilized
```

```
entity tx_register_decode is
    Port ( usb_mem_addr : in std_logic_vector(15 downto 0);
           data_load : in std_logic;
          pcie_copy_en : in std_logic;
          narrow_write : out std_logic_vector(8 downto 0);
          wide_write : out std_logic_vector(7 downto 0)
          );
end tx_register_decode;

architecture Behavioral of tx_register_decode is

begin

process(usb_mem_addr, data_load)
  begin
  if( (usb_mem_addr(15 downto 8) = "00000000") and (data_load = '1') )then
    case (usb_mem_addr(7 downto 4)) is
     when "0100" => narrow_write <= "000000001";
     when "0101" => narrow_write <= "000000010";
     when "0110" => narrow_write <= "000000100";
     when "0111" => narrow_write <= "000001000";
     when "1000" => narrow_write <= "000010000";
     when "1010" => narrow_write <= "000100000";
     when "1011" => narrow_write <= "001000000";
     when "1100" => narrow_write <= "010000000";
     when "1101" => narrow_write <= "100000000";
     when others => narrow_write <= "000000000";
    end case;
  else
    narrow_write <= "000000000";
   end if;
end process;

process(usb_mem_addr, pcie_copy_en, data_load)
  begin

  if( (usb_mem_addr(13) = '1') and (pcie_copy_en = '1') )then
    case (usb_mem_addr(4 downto 2)) is
     when "000" => wide_write <= "00000001";
     when "001" => wide_write <= "00000010";
     when "010" => wide_write <= "00000100";
     when "011" => wide_write <= "00001000";
     when "100" => wide_write <= "00010000";
     when "101" => wide_write <= "00100000";
     when "110" => wide_write <= "01000000";
     when "111" => wide_write <= "10000000";
     when others => wide_write <= "00000000";
    end case;
  else
    wide_write <= "00000000";
   end if;

end process;

end Behavioral;
```

tx_write_decoder.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

--  Uncomment the following lines to use the declarations that are
--  provided for instantiating Xilinx primitive components.
--library UNISIM;
--use UNISIM.VComponents.all;

entity tx_write_decoder is
```

```
    Port ( write_en : in std_logic;
        sel : in std_logic_vector(15 downto 0);
        write_en_out : out std_logic_vector(9 downto 0)
      );
end tx_write_decoder;

architecture Behavioral of tx_write_decoder is

begin

process(sel(15 downto 11),write_en)
begin
  if(write_en = '1') then
     case sel(15 downto 11) is
        when "00000"=>
        write_en_out <= "0000000001";  -- base
        when "00001" =>
        write_en_out <= "0000000010";  -- tx1
        when "00010" =>
        write_en_out <= "0000000100";  -- tx2
        when "00011" =>
        write_en_out <= "0000001000";  -- tx3
        when "00100" =>
        write_en_out <= "0000010000";  -- tx4
        when "00101" =>
        write_en_out <= "0000100000";  -- tx5
        when "00110" =>
        write_en_out <= "0001000000";  -- tx6
        when "00111" =>
        write_en_out <= "0010000000";  -- tx7
        when "01000" =>
        write_en_out <= "0100000000";  -- tx8
        when "01001" =>
        write_en_out <= "1000000000";  -- txclk
      when others =>
        write_en_out <= "0000000000";  -- do nothing
      end case;
    else
      write_en_out <= "0000000000";
    end if;
end process;

end Behavioral;
```

packet_framer.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;


-- Frame/Routing bits stored by packet and 'shaped' by byte counter
entity packet_framer is
    Port ( address : in STD_LOGIC_VECTOR (15 downto 0);
        data_in : in STD_LOGIC_VECTOR (7 downto 0);
        write_en : in STD_LOGIC;
           count : in  STD_LOGIC_VECTOR (7 downto 0);
           frame_ctl : in std_logic_vector(2 downto 0);
           regclk : in  STD_LOGIC;
        fastclk : in STD_LOGIC;
        frame : out STD_LOGIC_VECTOR (8 downto 0)
        );
end packet_framer;
```

```
architecture Behavioral of packet_framer is


signal frame_int : std_logic_vector(8 downto 0);

SUBTYPE header_width IS STD_LOGIC_VECTOR(8 DOWNTO 0);
-- Frame + 8 routing bits

TYPE header_length IS ARRAY (15 DOWNTO 0) OF header_width;
-- Currently supports 16 packets, extend as needed

SIGNAL hdr : header_length;

begin

process(address, data_in, write_en, regclk)
begin
-- technically should check bank bits == 00000 as well
  if( rising_edge(regclk) AND (write_en = '1') AND (address(8) = '1') ) then
  -- using bank 0, address 256+ to store this data.  this data is redundant to the
  -- block memory
    if (address(0) = '0') then
      hdr(conv_integer(address(4 downto 1)))(7 downto 0) <= data_in;
      -- low byte if addr(0) == 0
    else
      hdr(conv_integer(address(4 downto 1)))(8) <= data_in(0);
      -- just one bit if addr(0) == 1
    end if;
  end if;
end process;

process(frame_int, fastclk)
begin  -- register the data going out
  if( rising_edge(fastclk) ) then
    frame <= frame_int;
  end if;
end process;

process(count)
begin
  if (count(2 downto 0) = "111") then
    frame_int <= "000000000";  -- one cycle out of 8 the output MUST be 0 on all bits
  else  -- otherwise use the higher count bits as packet #, lookup stored values
    --frame_int <= "111111111" AND hdr(conv_integer(count(6 downto 3)));   -- AND isn't
needed
    frame_int <= hdr(conv_integer(count(6 downto 3)));
  end if;
end process;

end Behavioral;
```

rx_buffer_shift.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

--  Uncomment the following lines to use the declarations that are
--  provided for instantiating Xilinx primitive components.
--library UNISIM;
--use UNISIM.VComponents.all;

entity rx_buffer_shift is
    Port ( data_in : in std_logic_vector(7 downto 0);
           fastclk : in std_logic;
```

```
        clear : in std_logic;
        shift_en : in std_logic;
           read_addr : in std_logic_vector(5 downto 0);
        data_out : out std_logic_vector(7 downto 0)
        );
end rx_buffer_shift;

architecture Behavioral of rx_buffer_shift is

SUBTYPE sr_width IS STD_LOGIC_VECTOR(7 DOWNTO 0);
TYPE sr_length IS ARRAY (63 DOWNTO 0) OF sr_width;

SIGNAL sr : sr_length;

begin

process(fastclk,clear,shift_en,data_in)
begin
  if(clear = '1') then
    for i in 63 downto 0 loop
      sr(i) <= "00000000";
    end loop;
  elsif((rising_edge(fastclk)) AND (shift_en = '1')) then
    for i in 63 downto 1 loop
      sr(i) <= sr(i-1);
    end loop;
    sr(0) <= data_in;
  end if;
end process;

process(read_addr, sr)
begin
  data_out <= sr(conv_integer(read_addr));
end process;

end Behavioral;
```

Constraints File:

```
NET "SLOWCLOCK" CLOCK_DEDICATED_ROUTE = FALSE;
NET "PCLK_IN" CLOCK_DEDICATED_ROUTE = FALSE;
NET "RxPclkP" CLOCK_DEDICATED_ROUTE = FALSE;
NET "SLOWCLOCK" TNM_NET = SLOWCLOCK;
TIMESPEC TS_SLOWCLOCK = PERIOD "SLOWCLOCK" 24 MHz HIGH 50%;
NET "PCLK_IN" TNM_NET = PCLK_IN;
TIMESPEC TS_PCLK_IN = PERIOD "PCLK_IN" 325 MHz HIGH 50% INPUT_JITTER 200 ps;
NET "RxPclkP" TNM_NET = RxPclkP;
TIMESPEC TS_RxPclkP = PERIOD "RxPclkP" 325 MHz HIGH 50% INPUT_JITTER 200 ps;
NET "SYNC_OUT" MAXSKEW = 1.0 ns;
NET "RESET_READ" MAXSKEW = 0.5 ns;
```

APPENDIX C

USB COMMUNICATION INTERFACE

In this appendix, details of the interface(s) used to communicate with the test module from a host computer are presented. As discussed in Chapter 4, the test interface board designed for this research implements both a USB interface and a PCI Express link. Each interface can be used independently or conjointly, though for most day-to-day operations only the USB interface is utilized due to ease of use.

The Cypress microcontroller utilized to implement the electrical aspects of the USB interface utilized on the test development platform requires the compilation of a small selection of code that functions as a sort of firmware defining the specific operation of the unit and how it interacts with the computer and connected hardware element or the FPGA in this case. These files are compiled using a software package called Keil µVision and result in a core file that must be downloaded to the Cypress micontroller to be properly recognized by the interface software. Once programmed, the microcontroller accepts human interface device (HID) commands from a host pc and passes them on to the FPGA as sequenced bytes corresponding to address, data, and control information indicating if the operation should be interpreted as a write or read request (see Appendix A). The microcontroller code presented here is largely based on source code provided by John Hyde in USB Design-by-example [83], modified by Justin Davis, and further enhanced as part of this research. The files include:

- Hs.a51 – project definition
- Dtables.a51 – device declaration. String2, declared at the bottom of the file, allows software on the computer to identify the device and confirm that the microcontroller is programmed with a compatible configuration file

- Ezmain.a51 – primary function code. Expanded to handle 16-bit addresses and 8-bit data
- Ezint.a51 – device interrupt handler
- Decode.a51 – packet decode

Hs.A51

```
NAME    XilinxInterface
; Version 0.9
;
; Based on ButtonsAndLight example from USB-By-Example

;       g) EP0Size made an equate to ease coding of other components
EP0Size EQU     64      ; For EZ-USB

;
$INCLUDE(Declare.A51)
$INCLUDE(EZInt.A51)
$INCLUDE(EZMain.A51)
$INCLUDE(Decode.A51)
$INCLUDE(DTables.A51)


END
```

Dtables.a51

```
; This module declares the descriptors
;
; This example has one Device Descriptor with:
;  One Configuration - single IN port and single OUT port
;  One Interface - there is only one method of accessing the ports
;  One HID Descriptor - to make PC host software simpler
;  One Endpoint Descriptor - for HID Input Reports
;  One Report Descriptor - one byte IN and one byte OUT reports
;  Multiple Sting Descriptors - to aid the user
;
  CSEG
DeviceDescriptor:
  DB  18, 1      ; Length, Type
  DB  10H, 1     ; USB Rev 1.1 (=0110H, low=10H, High=01H)
  DB  0, 0, 0    ; Class, Subclass and Protocol
  DB  EP0Size
  DB  42H, 42H, 1, 42H, 0, 1; Vendor ID, Product ID and Version
  DB  1, 2, 0    ; Manufacturer, Product & Serial# Names
  DB  1          ; #Configs
ConfigurationDescriptor:
  DB  9, 2      ; Length, Type
  DB  LOW(ConfigLength), HIGH(ConfigLength)
  DB  1, 1, 0    ; #Interfaces, Configuration#, Config. Name
  DB  10000000b  ; Attributes = Bus Powered
  DB  250        ; Max. Power is 250x2 = 500mA
InterfaceDescriptor:
  DB  9, 4       ; Length, Type
  DB  0, 0, 1    ; No alternate setting, HID uses EP1
  DB  3          ; Class = Human Interface Device
  DB  0, 0       ; Subclass and Protocol
  DB  0          ; Interface Name
HIDDescriptor:
  DB  9, 21H     ; Length, Type
  DB  0, 1       ; HID Class Specification compliance
  DB  0          ; Country localization (=none)
  DB  1          ; Number of descriptors to follow
  DB  22H        ; And it's a Report descriptor
  DB  LOW(ReportLength), HIGH(ReportLength)
EndpointDescriptor:
  DB  7, 5         ; Length, Type
  DB  10000001b    ; Address = IN 1
  DB  00000011b    ; Interrupt
  DB  EP0Size, 0   ; Maximum packet size (this example only uses 1)
  DB  100          ; Poll every 0.1 seconds
ConfigLength  EQU $ - ConfigurationDescriptor

ReportDescriptor:   ; Generated with HID Tool, copied to here
  DB  6, 0, 0FFH    ; Usage_Page (Vendor Defined)
  DB  9, 1          ; Usage (I/O Device)
  DB  0A1H, 1       ; Collection (Application)
  DB  19H, 1        ;   Usage_Minimum (Button 1)
  DB  29H, 8        ;   Usage_Maximum (Button 8)
  DB  15H, 0        ;   Logical_Minimum (0)
  DB  25H, 1        ;   Logical_Maximum (1)
  DB  75H, 1        ;   Report_Size (1)
  DB  95H, 32       ;   Report_Count (8)
  DB  81H, 2        ;   Input (Data,Var,Abs)
  DB  19H, 1        ;   Usage_Minimum (Led 1)
  DB  29H, 24       ;   Usage_Maximum (Led 8)
  DB  91H, 2        ;   Output (Data,Var,Abs)
  DB  0C0H          ; End_Collection
ReportLength  EQU $-ReportDescriptor

String0:            ; Declare the UNICODE strings
  DB  4, 3, 9, 4    ; Only English language strings supported
String1:            ; Manufacturer
  DB  (String2-String1),3 ; Length, Type
  DB  "U",0,"S",0,"B",0," ",0,"D",0,"e",0,"s",0,"i",0,"g",0,"n",0," ", 0
  DB  "B",0,"y",0," ",0,"E",0,"x",0,"a",0,"m",0,"p",0,"l",0,"e",0
```

```
String2:      ; Product Name
  DB  (EndOfDescriptors-String2),3
  DB  "G",0,"e",0,"o",0,"r",0,"g",0,"i",0,"a",0," ",0,"T",0,"e",0,"c",0,"h",0," ",0
  DB  "T",0,"e",0,"s",0,"t",0," ",0,"C",0,"o",0,"r",0,"e",0," ",0,"V",0,"e",0,"r",0,"2",0
EndOfDescriptors:
  DB  0    ; Backstop for String Descriptors
```

Ezmain.a51

```
; This module initializes the microcontroller then executes MAIN forever
; It is hardware dependant

Reset:
 MOV SP, #0DFH        ; Initialize the Stack
 MOV PageReg, #7FH    ; Allows MOVX Ri to access EZ-USB memory

 MOV R0, #Low(USBControl) ; Simulate a disconnect
 MOVX A, @R0
 ANL A, #11110011b        ; Clear DISCON, DISCOE
 MOVX @R0, A
 CALL Wait100msec         ; Give the host time to react
 MOVX A, @R0              ; Reconnect with this new identity
 ORL A, #00000110b        ; Set DISCOE to enable pullup resistor
 MOVX @R0, A              ; Set RENUM so that 8051 handles USB requests
 CLR A
 MOV FLAGS, A             ; Start in Default state
InitVariables:
 MOV INControl, A
 MOV INAddressA, A
 MOV INAddressB, A
 MOV OUTAddressA, A
 MOV OUTAddressB, A
 MOV INData, A
 MOV OUTData, A
 MOV ValidCount, A
Initialize4msecCounter:
 MOV Msec_counter, A
InitializeIOSystem:         ; A=output, B=output C=input
 MOV R0, #LOW(PortA_Config) ; PageReg = 7F = HIGH(PortA_Config)
 CLR A
 MOVX @R0, A   ; No alternate functions on PortA
 INC R0
 MOVX @R0, A   ; No alternate functions on PortB
 INC R0
 MOVX @R0, A   ; No alternate functions on PortC

 MOV R1, #LOW(PortA_OE)
 CPL A          ; = 0FFH
 MOVX @R1, A    ; Enable PortA for Output
 INC R1         ; Point to PortB_OE
 MOVX @R1, A    ; Enable PortB for Output
 INC R1         ; Point to PortC_OE
 CLR A
 MOVX @R1, A    ; Enable Port C for Input

InitializeInterruptSystem:  ; First initialize the USB level
 MOV A, #00000001b
 MOV R0, #LOW(IN07IEN)
 MOVX @R0, A      ; Enable interrupts from EP0IN only
 INC R0
 CLR A
 MOVX @R0, A      ; Disable interrupts from OUT Endpoints 0-7
 INC R0
 MOV A, #00000011b
 MOVX @R0, A      ; Enable (Resume, Suspend,) SOF and SUDAV INTs
 INC R0
 MOV A, #00000001b
 MOVX @R0, A      ; Enable Auto Vectoring for USB interrupts
                  ; Now enable the main level
 MOV EIE, #00000001b        ; Enable INT2 = USB Interrupt (only)
 MOV EI, #10010000b         ; Enable interrupt subsystem (and Ser0 for dScope)

; Initialization Complete.
;
MAIN:
  NOP           ; Not much of a main loop for this example
  JMP MAIN      ; All actions are initiated by interrupts
```

```
; We are a slave, we wait to be told what to do

Wait100msec:
  MOV R7, #100
Wait1msec:          ; A delay loop
  MOV DPS, #0       ; Select primary DPTR
  MOV DPTR, #-1200
More: INC DPTR      ; 3 cycles
  MOV A, DPL        ; + 2
  ORL A, DPH        ; + 2
  JNZ More          ; + 3 = 10 cycles x 1200 = 1msec
  DJNZ  R7, Wait1msec
  RET

ProcessOutputReport:     ; A Report has just been received
; The report is four bytes long (Control, AddressA, AddressB, Data)
; Extended from J. Davis' implemenation which was only three bytes long
 MOV DPTR, #EP0OutBuffer ; Point to the Report
 MOVX A, @DPTR      ; Get Byte
 MOV INControl,A    ; Move it into memory as Control
 INC DPTR
 MOVX A,@DPTR       ; Get Byte
 MOV INAddressA, A  ; Move it into memory as AddressA

 INC DPTR
 MOVX A,@DPTR       ; Get Byte
 MOV INAddressB, A  ; Move it into memory as AddressB

 INC DPTR
 MOVX A,@DPTR       ; Get Byte
 MOV INData, A      ; Move it into memory as Data

 MOV A, INControl
 JZ ReadfromXilinx

WritetoXilinx:
; Write Address first on PortB, pulsing Write Bit
; Write Data next on PortB, pulsing Write Bit
 MOV A, INAddressA
 MOV DPTR, #PortB_Out
 MOVX @DPTR, A  ; Send the Address to Xilinx
 MOV     A, #16
 MOV     DPTR, #PortA_Out
 MOVX    @DPTR, A  ; Trigger set PortA4  (Set Write Addr Low Byte)

 CLR     A
 MOV     DPTR, #PortA_Out
 MOVX    @DPTR, A  ; Clear control bits

 MOV A, INAddressB
 MOV DPTR, #PortB_Out
 MOVX @DPTR, A  ; Send the Address to Xilinx
 MOV     A, #48
 MOV     DPTR, #PortA_Out
 MOVX    @DPTR, A  ; Trigger set both bits (Set Write Addr High Byte)

 CLR     A
 MOV     DPTR, #PortA_Out
 MOVX    @DPTR, A  ; Clear control bits

 MOV A, INData
 MOV DPTR, #PortB_Out
 MOVX @DPTR, A  ; Send the Data to Xilinx
 MOV     A, #32
 MOV     DPTR, #PortA_Out
 MOVX    @DPTR, A  ; Trigger set PortA5  (Set Write Data & Clear Write Addr)
 CLR     A
 MOVX    @DPTR, A  ; Trigger clear PortA5  (Clear Write Data)
 RET

ReadfromXilinx:
```

```
; Write Address first on PortB, pulsing Read Bit
; Read Data next on PortC
 MOV A, INAddressA
 MOV OUTAddressA, A
 MOV DPTR, #PortB_Out
 MOVX @DPTR, A  ; Send the Address to Xilinx
 MOV     A, #16
 MOV     DPTR, #PortA_Out
 MOVX    @DPTR, A  ; Trigger set PortA5  (Set Load Addr)

 CLR     A
 MOVX    @DPTR, A  ; Trigger clear PortA5  (Clear Load Addr)

 MOV A, INAddressB
 MOV OUTAddressB, A
 MOV DPTR, #PortB_Out
 MOVX @DPTR, A  ; Send the Address to Xilinx
 MOV     A, #48
 MOV     DPTR, #PortA_Out
 MOVX    @DPTR, A  ; Trigger set PortA5  (Set Load Addr)

 CLR     A
 MOVX    @DPTR, A  ; Trigger clear PortA5  (Clear Load Addr)

 MOV DPTR, #PortC_Pins
 MOVX A, @DPTR
 MOV OUTData, A  ; Read Data from Xilinx
 MOV A, ValidCount
 INC A
 MOV ValidCount, A

CreateInputReport:   ; Called when data is requested by Host
; The report is 4 bytes: Valid Byte, Address Low, Address High, Data
; Value in A is Valid Byte (leftover from above)
 MOV DPTR, #EP1InBuffer ; Point to the buffer
 MOVX @DPTR, A  ; Ready Valid Byte
 INC DPTR   ; increment the buffer

 MOV A, OUTAddressA
 MOVX @DPTR, A  ; Ready Address
 INC DPTR   ; increment the buffer

 MOV A, OUTAddressB
 MOVX @DPTR, A  ; Ready Data
 INC DPTR   ;increment the buffer

 MOV A, OUTData
 MOVX @DPTR, A  ; Ready Data
 INC DPTR   ;increment the buffer

 MOV DPTR, #IN1ByteCount
 MOV A, #4   ; 4 total bytes now
 MOVX @DPTR, A  ; Endpoint 1 now 'armed', next IN will get data
 RET
```

Ezint.a51

```
; This module contains all the EZUSB-specific hardware code
; This module also contains all of the interrupt vector declarations and
; the first level interrupt servicing (register save, call subroutine,
; clear interrupt source, restore registers, return)
; Suspend and Resume are handled totally in this module
;
; A Reset sends us to Program space location 0
  CSEG AT 0   ; Code space
  USING 0     ; Reset forces Register Bank 0
  LJMP  Reset
;
; The interrupt vector table is also located here
; EZ-USB has two levels of USB interrupts:
; 1-the main level is described in this table (at ORG 43H)
; 2-there are 21 sources of USB interrupts and these are described in USB_ISR
; This means that two levels of acknowledgement and clearing will be required
; LJMP  INT0_ISR  ; Features not used are commented out
; ORG 0BH
; LJMP  Timer0_ISR
; ORG 13H
; LJMP  INT1_ISR
; ORG 1BH
; LJMP  Timer1_ISR
; ORG 23H
; LJMP  UART0_ISR
; ORG 2BH
; LJMP  Timer2_ISR
; ORG 33H
; LJMP  WakeUp_ISR
; ORG 3BH
; LJMP  UART1_ISR
  ORG 43H
  LJMP  USB_ISR   ; Auto Vector will replace byte 45H
; ORG 4BH
; LJMP  I2C_ISR
; ORG 53H
; LJMP  INT4_ISR
; ORG 5BH
; LJMP  INT5_ISR
; ORG 63H
; LJMP  INT6_ISR

  ORG 1200H   ; Load above monSIO0.hex
USB_ISR:LJMP  SUDAV_ISR
        DB  0   ; Pad entries to 4 bytes
  LJMP  SOF_ISR
  DB  0
  LJMP  SUTOK_ISR
  DB  0
  LJMP  Suspend_ISR
  DB  0
  LJMP  USBReset_ISR
  DB  0
  LJMP  Reserved
  DB  0
  LJMP  EP0In_ISR
; DB  0               ; Comment out features not used
; LJMP  EP0Out_ISR
; DB  0
; LJMP  EP1In_ISR
; DB  0
; LJMP  EP1Out_ISR
; DB  0
; LJMP  EP2In_ISR
; DB  0
; LJMP  EP2Out_ISR
; DB  0
```

```
; LJMP  EP3In_ISR
; DB  0
; LJMP  EP3Out_ISR
; DB  0
; LJMP  EP4In_ISR
; DB  0
; LJMP  EP4Out_ISR
; DB  0
; LJMP  EP5In_ISR
; DB  0
; LJMP  EP5Out_ISR
; DB  0
; LJMP  EP6In_ISR
; DB  0
; LJMP  EP6Out_ISR
; DB  0
; LJMP  EP7In_ISR
; DB  0
; LJMP  EP7Out_ISR
; End of Interrupt Vector tables

; When a feature is used insert the required interrupt processing here
; The example use only used Endpoints 0 and 1 and also SOF for timing
Reserved:
INT0_ISR:
Timer0_ISR:
INT1_ISR:
Timer1_ISR:
UART0_ISR:
Timer2_ISR:
UART1_ISR:
I2C_ISR:
INT4_ISR:
INT5_ISR:
INT6_ISR:
SUTOK_ISR:
EP0Out_ISR:
EP1In_ISR:
EP1Out_ISR:
EP2In_ISR:
EP2Out_ISR:
EP3In_ISR:
EP3Out_ISR:
EP4In_ISR:
EP4Out_ISR:
EP5In_ISR:
EP5Out_ISR:
EP6In_ISR:
EP6Out_ISR:
EP7In_ISR :
EP7Out_ISR:
Not_Used:     ; Should not get any of these
  RETI

ClearINT2:      ; Tell the hardware that we're done
  MOV A, EXIF
  CLR ACC.4   ; Clear the Interrupt 2 bit
  MOV EXIF, A
  RET

USBReset_ISR:     ; Bus has been Reset, move to DEFAULT state
  CLR Configured
  CALL ClearINT2  ; No need to clear source of interrupt
  RETI

Suspend_ISR:    ; SIE detected an Idle bus
  MOV A, PCON
  ORL A, #1
  MOV PCON, A   ; Go to sleep!
  NOP
  NOP           ; Wake up here due to a USBResume
```

```
  NOP
  CALL  ClearINT2
  RETI

WakeUp_ISR:       ; Not using external WAKEUP in these examples
                  ; So this must be due to a USBResume
  CLR EICON.4     ; Clear the wakeup interrupt source
  RETI

EP0In_ISR:            ; A prepared packet has been read by PC host
  MOV A, SaveLength   ; Do I have any more data to send?
  JZ  NoMoreToSend
  MOV DPH, SaveDPH    ; Retreive descriptor pointer
  MOV DPL, SaveDPL
  CALL  SendNextPieceOfDescriptor
NoMoreToSend:
  CALL  ClearINT2
  MOV A, #00000001b
  MOV DPTR, #IN07IRQ
  MOVX  @DPTR, A    ; Clear source of interrupt
  RETI

SOF_ISR:        ; A Start-Of-Frame packet has been received
; This routine services the real time interrupt
; It is also responsible for the "real world" buttons and lights
;
ServiceTimerRoutine:
; LED routine moved to exmain.a51
  MOV A, ValidCount
  CALL  CreateInputReport

Done: CALL  ClearINT2
                          ; Clear the source of the interrupt
  MOV A, #00000010b
ExitISR:MOV DPTR, #USBIRQ
  MOVX  @DPTR, A
  RETI

SUDAV_ISR:              ; A Setup packet has been received
  MOV SaveLength, #0    ; Clear any pending transactions (if any)
  MOV DPTR, #SETUPDAT   ; Copy packet to direct access memory
  MOV R0, #SetupData
  MOV R7, #8
CopySD: MOVX  A, @DPTR
  MOV @R0, A
  INC DPTR
  INC R0
  DJNZ  R7, CopySD
  CALL  ServiceSetupPacket  ; Handle the decode of the Setup packet
; if SetAddress { Update SIE address }  // NOP on EZ-USB
; if STALL { Stall the endpoint }
; if SendData {
;   if IsDescriptor { send DPTR->descriptor, A = length }
; else { send ReplyBuffer }
; }
  JB  STALL, SendSTALL
  JNB SendData, HandShake
  JB  IsDescriptor, LoadEP0
                          ; Send data in ReplyBuffer
  MOV DPTR, #EP0InBuffer+1
  MOV R0, #ReplyBuffer+1
  MOV R7, #2              ; Copy the two byte buffer
CopyRB: MOV A, @R0
  MOVX  @DPTR, A
  DEC DPL
  DEC R0
  DJNZ  R7, CopyRB
  MOV A, @R0              ; Get BufferCount
SendEP0InBuffer:
  MOV DPTR, #In0ByteCount
StartXfer:
```

```
  MOVX  @DPTR, A            ; This write initiates the transfer
HandShake:                  ; Handshake with host
  MOV R7, #00000010b        ; Set HSNAK to tell the SIE that we're done
SetEP0Control:
  MOV DPTR, #EP0Control
  MOVX  A, @DPTR
  ORL A, R7
  MOVX  @DPTR, A            ; We're done
  CALL  ClearINT2
  MOV A, #00000001b         ; Clear the source of the interrupt
  JMP ExitISR
SendSTALL:                  ; Invalid Request was received
  MOV R7, #00000011b        ; Set EP0STALL and HSNAK
  JMP SetEP0Control
LoadEP0:                    ; Send the data pointed to by DPTR
  MOV R7, A                 ; Save LENGTH
; Need to return the smaller of "Requested Length" and "Actual Length"
; If "Requested Length" > 255 then use "Actual Length"
; There are no descriptors > 255 in this example
  MOV A, wLengthHigh
  JNZ UseActual
  CLR C
  SUBB  A, wLengthLow
  MOV A, wLengthLow         ; This does not affect Carry
  JNC UsewLengthLow
UseActual:
  MOV A, R7
UsewLengthLow:
SendNextPieceOfDescriptor:  ; DPTR -> Descriptor to be sent
  MOV R7, A                 ; Save LENGTH again
  MOV SaveLength, #0        ; Default case, overwrite if necessary
; Do I have more than a single packet to send?
  CLR C
  SUBB  A, #EP0Size
  JC  SendPacket
; Need to send multiple packets.
; Calculate and save address of next packet, send next packet now
  MOV SaveLength, A   ; Send these next time
  MOV R7, #EP0Size
  PUSH  DPH     ; Save current pointer
  PUSH  DPL
  MOV A, R7     ; Retreive length
  CALL  BumpDPTR
  MOV SaveDPH, DPH
  MOV SaveDPL, DPL
  POP DPL
  POP DPH
SendPacket:
  MOV A, R7     ; Retreive length
  MOV R6, A     ; Save length in R6 for move
  MOV R0, #LOW(EP0InBuffer) ; PageReg = 7FH = HIGH(EP0InBuffer)
CopySTD:MOVX  A, @DPTR
  MOVX  @R0, A
  INC DPTR
  INC R0
  DJNZ  R6, CopySTD
  MOV A, R7     ; Retrieve LENGTH
  JMP SendEP0InBuffer

GetOutputReport:      ; Wait for this, it's next on USB
  MOV DPTR, #Out0ByteCount  ; Enable EP0OutBuffer to receive data
  MOVX  @DPTR, A    ; Any value will do
        MOV DPTR, #EP0Control ; Wait for valid data in EP0OutBuffer
Wait40: MOVX  A, @DPTR
  ANL A, #00001000b   ; Check OUTBSY
  JNZ Wait40
  RET
```

Decode.a51

```
; This module is common to all of the examples.
; It decodes the USB Setup Packets and generates appropriate responses.
; Interpretation of Reports is handled by MAIN
;
  CSEG
ServiceSetupPacket:
  MOV A, RequestType
  MOV C, ACC.7    ; Bit 7 = 1 means IO device needs to send data to PC Host
  MOV SendData, C
  ANL A, #01011100b   ; IF RequestType[6.4.3.2] = 1 THEN goto BadRequest
  JNZ BadRequest
  MOV A, RequestType    ; IF RequestType[1&0] = 1 THEN goto BadRequest
  MOV C, ACC.0
  ANL C, ACC.1
  JC  BadRequest
  JNB ACC.5, NotB5    ; IF RequestType[5] = 1 THEN RequestType[1,0] = [1,1]
  MOV A, #00000011b
NotB5:  ANL A, #00000011b   ; Set CommandIndex[5,4] = RequestType[1,0]
  SWAP  A
  MOV R7, A     ; Save HI nibble of CommandIndex
          ; Set CommandIndex[3,0] = Request[3,0]
  MOV A, Request
  ANL A, #11110000b   ; Check if Request > 15
  JNZ BadRequest
  MOV A, Request
  ANL A, #00001111b   ; Only 13 are defined today, handle in table
  ORL A, R7
; CALL  CorrectSubroutine ; goto CommandTable(CommandIndex)
CorrectSubroutine:      ; Jump to the subroutine that DPTR is pointing to
  MOV ReplyCount, #1    ; Set up a default reply
  MOV ReplyBuffer, #0
  MOV ReplyBuffer+1, #0
  CLR SetAddress    ; Clear all flags
  CLR STALL
  CLR IsDescriptor
  MOV DPTR, #CommandTable
  CALL  BumpDPTR    ; Point to entry
  MOVX  A, @DPTR    ; Get the offset
  MOV DPTR, #Subroutines
  JMP   @A+DPTR     ; Go to the correct Subroutine

BadRequest:       ; Decoded a Bad Request, STALL the Endpoint
  SETB  STALL
  RET
          ; Support routines
NextDPTR:       ; Returns (DPTR + byte DPTR is pointing to)
  MOVX  A, @DPTR
BumpDPTR:       ; Returns (DPTR + ACC)
  ADD A, DPL
  MOV DPL, A
  JNC Skip
  INC DPH     ; Need 16 bit arithmetic here
Skip: RET

; Since the table only contains byte offsets, it is important that all these routines are
; within one page (100H) of Subroutines
; V3.0 - CommandTable moved outside of this one page limited space
CommandTable:
; First 16 commands are for the Device
  DB LOW(Device_Get_Status - Subroutines)
  DB LOW(Device_Clear_Feature - Subroutines)
  DB LOW(Invalid - Subroutines)
  DB LOW(Device_Set_Feature - Subroutines)
  DB LOW(Invalid - Subroutines)
  DB LOW(Set_Address - Subroutines)
  DB LOW(Get_Descriptor - Subroutines)
  DB LOW(Set_Descriptor - Subroutines)
  DB LOW(Get_Configuration - Subroutines)
  DB LOW(Set_Configuration - Subroutines)
```

```
  DB LOW(Invalid - Subroutines)
  DB LOW(Invalid - Subroutines)
  DB LOW(Invalid - Subroutines)
  DB LOW(Invalid - Subroutines)
  DB LOW(Invalid - Subroutines)
  DB LOW(Invalid - Subroutines)
; Next 16 commands are for the Interface
  DB LOW(Interface_Get_Status - Subroutines)
  DB LOW(Interface_Clear_Feature - Subroutines)
  DB LOW(Invalid - Subroutines)
  DB LOW(Interface_Set_Feature - Subroutines)
  DB LOW(Invalid - Subroutines)
  DB LOW(Invalid - Subroutines)
  DB LOW(Get_Class_Descriptor - Subroutines)
  DB LOW(Set_Class_Descriptor - Subroutines)
  DB LOW(Invalid - Subroutines)
  DB LOW(Invalid - Subroutines)
  DB LOW(Get_Interface - Subroutines)
  DB LOW(Set_Interface - Subroutines)
  DB LOW(Invalid - Subroutines)
  DB LOW(Invalid - Subroutines)
  DB LOW(Invalid - Subroutines)
  DB LOW(Invalid - Subroutines)
; Next 16 commands are for the Endpoint
  DB LOW(Endpoint_Get_Status - Subroutines)
  DB LOW(Endpoint_Clear_Feature - Subroutines)
  DB LOW(Invalid - Subroutines)
  DB LOW(Endpoint_Set_Feature - Subroutines)
  DB LOW(Invalid - Subroutines)
  DB LOW(Invalid - Subroutines)
  DB LOW(Invalid - Subroutines)
  DB LOW(Invalid - Subroutines)
  DB LOW(Invalid - Subroutines)
  DB LOW(Invalid - Subroutines)
  DB LOW(Invalid - Subroutines)
  DB LOW(Invalid - Subroutines)
  DB LOW(Endpoint_Sync_Frame - Subroutines)
  DB LOW(Invalid - Subroutines)
  DB LOW(Invalid - Subroutines)
  DB LOW(Invalid - Subroutines)
; Next 16 commands are Class Requests
  DB LOW(Invalid - Subroutines)
  DB LOW(Get_Report - Subroutines)
  DB LOW(Get_Idle - Subroutines)
  DB LOW(Get_Protocol - Subroutines)
  DB LOW(Invalid - Subroutines)
  DB LOW(Invalid - Subroutines)
  DB LOW(Invalid - Subroutines)
  DB LOW(Invalid - Subroutines)
  DB LOW(Invalid - Subroutines)
  DB LOW(Set_Report - Subroutines)
  DB LOW(Set_Idle - Subroutines)
  DB LOW(Set_Protocol - Subroutines)
  DB LOW(Invalid - Subroutines)
  DB LOW(Invalid - Subroutines)
  DB LOW(Invalid - Subroutines)
  DB LOW(Invalid - Subroutines)

Subroutines:
;
; Many requests are INVALID for this example
Get_Protocol:     ; We are not a Boot device
Set_Protocol:       ; We are not a Boot device
Set_Descriptor:     ; Our Descriptors are static
Set_Class_Descriptor:           ; Our Descriptors are static
Set_Interface:      ; We only have one Interface
Get_Interface:      ; We do not have an Alternate setting
Set_Idle:     ; V3.0 Optional command, not supported
Get_Idle:     ; V3.0 Optional command, not supported
Device_Set_Feature:     ; We have no features that can be set or cleared
Interface_Set_Feature:    ; We have no features that can be set or cleared
```

```
Endpoint_Set_Feature:     ; We have no features that can be set or cleared
Endpoint_Clear_Feature:   ; V3.0 We have no features that can be set or cleared
Device_Clear_Feature:   ; We have no features that can be set or cleared
Interface_Clear_Feature:  ; We have no features that can be set or cleared
Endpoint_Sync_Frame:    ; We are not an Isonchronous device

Invalid:      ; Invalid Request made, STALL the Endpoint
  SETB  STALL
Reply:  RET

Set_Address:      ; Set the address that the SIE will respond to
  SETB  SetAddress
  RET

Set_Report:       ; Host wants to sent us a Report.
; The ONLY case in this example where host sends data to us
  JNB Configured, Invalid ; Need to be Configured to do this command
  CALL  GetOutputReport   ; Handled in EZUSB.A51
  JMP ProcessOutputReport ; RETurn via this subroutine
Get_Report:       ; Host wants a Report
  JNB Configured, Invalid ; Need to be Configured to do this command
  MOV ReplyBuffer, #42H   ; Reply with a recognizable (arbitary) value
  RET
Get_Configuration:      ; Respond with CurrentConfiguration
  MOV ReplyBuffer, CurrentConfiguration
  RET
Device_Get_Status:      ; Only two bits of Device Status are defined
  MOV ReplyBuffer, #1   ; Bit 1=Remote Wakeup(=0), Bit 0=Self Powered(=1)
  RET
Interface_Get_Status:     ; Interface Status is currently defined as 0
Endpoint_Get_Status:
        MOV ReplyCount, #2    ; Need a two byte 0 response
  RET
Set_Configuration:        ; Valid values are 0 and 1
  MOV A, wValueLow
  JZ  Deconfigured
  DEC A
  JNZ Invalid
  SETB  Configured
  MOV CurrentConfiguration, #1
  RET
Deconfigured:
  CLR Configured
  MOV CurrentConfiguration, A
  RET
Get_Descriptor:         ; Host wants to know who/what we are
  SETB  IsDescriptor
  MOV A, wValueHigh
  DEC A     ; Valid Values are 1, 2 and 3
  MOV DPTR, #DeviceDescriptor
  JZ  ReturnLength
  DEC A
  MOV DPTR, #ConfigurationDescriptor
  JNZ TryString
  MOV A, #ConfigLength
  RET
TryString:
  DEC A
  JNZ Invalid
; Request is for a String Descriptor
  MOV DPTR, #String0    ; Point to String 0
  MOV A, wValueLow    ; Get String Index
NextString:
  JZ  ReturnLength
  MOV R7, A     ; Save String Index
  CALL  NextDPTR
  MOVX  A, @DPTR    ; Get the String Length (= 0 means we're at Backstop)
  JZ  Invalid     ; Asked for a string I don't have
  MOV A, R7
  DEC A
  JMP NextString    ; Check if we are there yet
```

```
Get_Class_Descriptor:       ; Valid values are 21H, 22H, 23H for Class Request
  SETB  IsDescriptor
  MOV A, wValueHigh
  CLR C
  SUBB  A, #21H
  MOV DPTR, #HIDDescriptor
  JZ  ReturnLength
  DEC A
  MOV DPTR, #ReportDescriptor
  JZ  ReturnRDlength
; DEC A     ; This example does not use Physical Descriptors
; JZ  Send_Physical_Descriptor
  JMP Invalid
;
ReturnLength:
  MOVX  A, @DPTR    ; Get Descriptor Length (first byte)
  RET
ReturnRDlength:       ; Report Descriptor is different format
  MOV A, #ReportLength
  RET
; Error check: this MUST be on within a page of Subroutines
WithinSamePage EQU $ - Subroutines
;
```

Declare.a51

```
; This module declares the variables and constants used in the examples
; It is common to all of the examples
;
; Declare Special Function Registers used
TimerControl  DATA  088H
TimerMode     DATA  089H
Timer0High    DATA  08CH
EI            DATA  0A8H
EIE           DATA  0E8H  ; EZ-USB specific
EXIF          DATA  091H  ; EZ-USB specific
EICON         DATA  0D8H  ; EZ-USB specific
PageReg       DATA  092H  ; EZ-USB specific, used with MOVX @Ri
DPS           DATA  086H  ; EZ-USB specific, used with dual data pointers
;
; "External" memory locations used, EZ-USB specific
; Note that most of these variables are in Page 7FH
SETUPDAT      EQU 07FE8H
SUDPTR        EQU 07FD4H
EP0Control    EQU 07FB4H
EP0InBuffer   EQU 07F00H
EP0OutBuffer  EQU 07EC0H    ; Not in Page 7FH
EP1InBuffer   EQU 07E80H    ; Not in Page 7FH
IN0ByteCount  EQU 07FB5H
Out0ByteCount EQU 07FC5H
IN1ByteCount  EQU 07FB7H
IN07IEN       EQU 07FACH
IN07IRQ       EQU 07FA9H
OUT07IEN      EQU 07FADH
OUT07IRQ      EQU 07FAAH
USBIEN        EQU 07FAEH
USBIRQ        EQU 07FABH
USBControl    EQU 07FD6H
I2CData       EQU 07FA6H
I2CControl    EQU 07FA5H
PortA_Config  EQU 07F93H
PortB_Config  EQU 07F94H
PortC_Config  EQU 07F95H
PortA_OUT     EQU 07F96H
PortB_OUT     EQU 07F97H
PortC_OUT     EQU 07F98H
PortA_PINS    EQU 07F99H
PortB_PINS    EQU 07F9AH
PortC_PINS    EQU 07F9BH
PortA_OE      EQU 07F9CH
PortB_OE      EQU 07F9DH
PortC_OE      EQU 07F9EH
;
; Byte Variables

    DSEG  AT 20H
FLAGS:    DS  1 ; This register is bit-addressable
; Bit Variables
Configured  EQU FLAGS.0 ; Is this device configured
STALL   EQU FLAGS.1 ; Need to STALL endpoint 0
SendData  EQU FLAGS.2 ; Need to send data to PC Host
IsDescriptor  EQU FLAGS.3 ; Enable a shortcut reply
SetAddress  EQU FLAGS.4 ; Set the SIE address
;
MonitorSpace: DS  1FH ; Used by Dscope
;Expired_Time:  DS  1 ; A downcounter for timed Reports
ReplyCount: DS  1 ; Byte count for following buffer
ReplyBuffer:  DS  2 ; Buffer for immediate reply
CurrentConfiguration:
    DS  1 ; Some examples support > 1 configurations
SaveDPH:  DS  1 ; Needed to save Descriptor Pointer ..
SaveDPL:  DS  1 ; .. for descriptors > EP0Size
SaveLength: DS  1 ; Number of bytes still to send
SetupData:      ; Buffer in direct access memory
RequestType:  DS  1
```

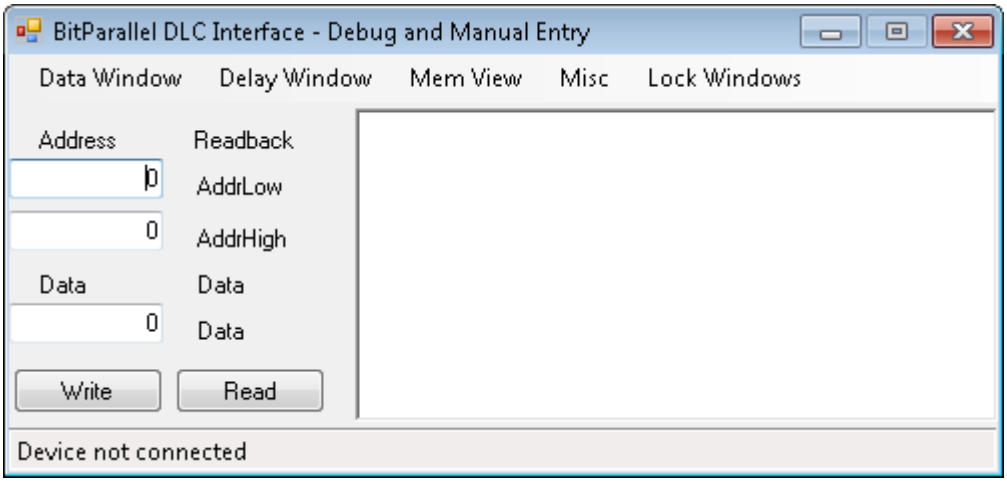
```
Request:  DS  1
wValueLow:  DS  1
wValueHigh: DS  1
wIndexLow:  DS  1
wIndexHigh: DS  1
wLengthLow: DS  1
wLengthHigh:  DS  1
;
;Old_Buttons: DS  1 ; Used by BAL: stores current button position
;LEDstrobe: DS  1 ; Used by BAL: strobe one LED on at a time
;LEDvalue:  DS  1 ; Used by BAL: stores current LED value
Msec_Counter: DS  1 ; Used by BAL: counts up to 4 msec
INAddressA: DS  1 ; Incoming Address from USB, lower byte
INAddressB: DS  1 ; Incoming Address from USB, upper byte
OUTAddressA:  DS  1 ; Outgoing Address to USB, lower byte
OUTAddressB:  DS  1 ; Outgoing Address to USB, upper byte
INData:   DS  1 ; Incoming Data from USB
OUTData:  DS  1 ; Outgoing Data to USb
INControl:  DS  1 ; Control Byte containing Read/Write Info
ValidCount: DS  1 ; Keeps count of valid outputs
;
```

APPENDIX D

HOST COMPUTER SOFTWARE

This section provides implementation details of the host computer software which is designed to interact with the user to control the high level aspects of the test system as well as allow low-level access to the memory elements inside the FPGA to facilitate design and debug efforts. The program is written in Visual C# which is part of the Microsoft Visual Studio programming suite.

The code below is built upon and expanded from the Human Interaction Device generic_hid_cs code base as originally created by Jan Alexson and available at http://www.lvr.com/hidpage.htm.



This is the primary interface window for the software. This window implements the majority of the device level code, detecting and interacting with the HID at the byte level. This module implements two important functions, SimpleSend and SimpleRead which can be used to send and receive data to and from the test platform without understanding the specifics of the underlying code. Where possible, as many of the read and write operations are performed with these two functions. However, some older sections of code, preceding the creation of these functions, are implemented using the lower level function calls.

This window provides high address, low address, and data entry boxes with a pair of buttons for writing and reading. Values are entered into these boxes in integers and can be used for system debugging. Feedback information is provided through the various message labels in the window and the text field at the bottom right. Additional function windows can be accessed through the menu bar at the top.

BitParallel.cs

```
using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
```

```
using System.Text;
using System.Windows.Forms;
using Microsoft.Win32.SafeHandles;
using System.Runtime.InteropServices;
using System.Diagnostics;
using System.Threading;
using Microsoft.VisualBasic;

namespace BitParallel_HID_Interface
{
  internal enum reportType { Read, Write };

  public partial class BitParallel : Form
  {
    private Byte[] dummyReport = new Byte[5];
    private Byte validByte = 0;

    private IntPtr deviceNotificationHandle;
    private SafeFileHandle hidHandle;
    private String hidUsage;
    private Boolean myDeviceDetected;
    private String myDevicePathName;
    private SafeFileHandle readHandle;
    private SafeFileHandle writeHandle;

    private Boolean exclusiveAccess;

    private DelayWindow frameDelay;
    private DataWindow frameData;
    private MemoryPanel frameMemory;

    private Debugging MyDebugging = new Debugging(); //  For viewing
results of API calls via Debug.Write.
    private DeviceManagement MyDeviceManagement = new
DeviceManagement();
    internal Hid MyHid = new Hid();

    public Boolean emulate = false;


    public BitParallel()
    {
      InitializeComponent();
    }

    private void BitParallel_Load(object sender, EventArgs e)
    {
      FindTheHid();
      frameDelay = new DelayWindow(this);
      frameData = new DataWindow(this);
      frameMemory = new MemoryPanel(this);
      frameDelay.Visible = false;
      frameData.Visible = false;
      frameDelay.Owner = this;
      frameData.Owner = this;
      frameMemory.Visible = false;
      frameMemory.Owner = this;
```

```
      if (myDeviceDetected == true)
      {
        Boolean successRead = false;
        successRead = ReadInputReport(ref dummyReport);
        if (successRead == true)
        {
          validByte = dummyReport[1];
          labelValid.Text = validByte.ToString();
        }
      }

      timer1.Enabled = false;
    }

    private void buttonWrite_Click(object sender, EventArgs e)
    {
      String byteValue = null;
      Int32 count = 0;
      Boolean success = false;
      Byte[] outputReportBuffer = null;

      if (MyHid.Capabilities.OutputReportByteLength > 3)
      {
        outputReportBuffer = new
Byte[MyHid.Capabilities.OutputReportByteLength];

        //  Store the report ID in the first byte of the buffer:
        outputReportBuffer[0] = 0; // report ID

        outputReportBuffer[1] = Convert.ToByte(reportType.Write);

        outputReportBuffer[2] = Convert.ToByte(textBoxAddrLow.Text);

        if (MyHid.Capabilities.OutputReportByteLength == 5)
        {

          outputReportBuffer[3] = Convert.ToByte(textBoxAddrHigh.Text);
          outputReportBuffer[4] = Convert.ToByte(textBoxData.Text);  //
double byte addressing detected, include high address and then data
        }
        else
        {
          outputReportBuffer[3] = Convert.ToByte(textBoxData.Text);  //
single byte addressing detected, ignore high address
        }
      }
      success = SendOutputReport(ref outputReportBuffer);

      if (success)
      {
        DebugLine("An Output report has been written.");

        //  Display the report data in the form's list box.
        DebugLine(" Output Report ID: " + String.Format("{0:X2} ",
outputReportBuffer[0]));
        DebugLine(" Output Report Data:");
```

```
        for (count = 0; count <= outputReportBuffer.Length - 1;
count++)
        {
          //  Display bytes as 2-character hex strings.
          byteValue = String.Format("{0:X2} ",
outputReportBuffer[count]);
          DebugLine(byteValue);
        }
      }
      else
      {
        DebugLine("The attempt to write an Output report has failed.");
      }
    }

    private void buttonRead_Click(object sender, EventArgs e)
    {
      Boolean successWrite = false;
      Byte[] outputReportBuffer = new
Byte[MyHid.Capabilities.OutputReportByteLength];

      if (MyHid.Capabilities.OutputReportByteLength > 3)
      {
        outputReportBuffer[0] = 0; // reportID
        outputReportBuffer[1] = Convert.ToByte(reportType.Read);
        outputReportBuffer[2] = Convert.ToByte(textBoxAddrLow.Text);

        if (MyHid.Capabilities.OutputReportByteLength == 5)
        {

          outputReportBuffer[3] = Convert.ToByte(textBoxAddrHigh.Text);
          outputReportBuffer[4] = Convert.ToByte(textBoxData.Text);  //
double byte addressing detected, include high address and then data
        }
        else
        {
          outputReportBuffer[3] = Convert.ToByte(textBoxData.Text);  //
single byte addressing detected, ignore high address
        }
      }
      successWrite = SendOutputReport(ref outputReportBuffer);
      if (successWrite == true)
      {
        DebugLine("Read address written.  Waiting for result");
        validByte++;
        int retry = 0;
        while (retry < 128)
        {
          ReadInputReport(ref dummyReport);
          if (dummyReport[1] == validByte)
          {
            DebugLine("Read success: " + dummyReport[4]);
            retry = 128;
          }
          else
          {
            if (menuItemSilenceOverride.Checked)
```

```
              DebugLine("Saw unexpected byte on try " + retry + ": " +
dummyReport[1] + ".  Expected: " + validByte);
            Thread.Sleep(10);
            retry++;
          }
        }

      }

    }

    internal Boolean SendOutputReport(ref Byte[] outputReportBuffer)
    {
      Boolean success = false;

      if (myDeviceDetected == false)
        FindTheHid();

      if (myDeviceDetected == false)
        return false;

      //debugText.Focus();

      try
      {
        //  Don't attempt to exchange reports if valid handles aren't
available
        //  (as for a mouse or keyboard under Windows 2000/XP.)
        if (!readHandle.IsInvalid && !writeHandle.IsInvalid)
        {
          //  Don't attempt to send an Output report if the HID Output
report is too small.
          if (MyHid.Capabilities.OutputReportByteLength > 3)
          {
            //  Write a report.
            Hid.OutputReportViaInterruptTransfer myOutputReport = new
Hid.OutputReportViaInterruptTransfer();
            success = myOutputReport.Write(outputReportBuffer,
writeHandle);
          }
          else
          {
            DebugLine("The HID doesn't have an Output report or it's
too small (" + MyHid.Capabilities.OutputReportByteLength + ").");
          }
        }
        else
        {
          DebugLine("Invalid handle. The device is probably a system
mouse or keyboard.");
          DebugLine("No attempt to write an Output report or read an
Input report was made.");
        }
      }
      catch (Exception ex)
      {
        throw;
```

```
      }

      return success;
    }

    internal Boolean ReadInputReport(ref Byte[] inputReportBuffer)
    {
      Boolean success = false;

      if (myDeviceDetected == false)
        FindTheHid();

      if (myDeviceDetected == false)
        return false;

      try
      {
        Hid.InputReportViaInterruptTransfer myInputReport = new
Hid.InputReportViaInterruptTransfer();
        myInputReport.Read(hidHandle, readHandle, writeHandle, ref
myDeviceDetected, ref inputReportBuffer, ref success);
      }
      catch (Exception ex)
      {
        throw;
      }

      return success;
    }

    internal Boolean SimpleSend(byte lowerAddr, byte upperAddr, byte
data, Boolean silent)
    {
      Boolean success = false;
      int dataLoc = 3;

      silent = silent & !menuItemSilenceOverride.Checked;
      //if (silent == false)
      //debugText.Focus();  // only give the window focus if we need to
output something

      Byte[] outputReportBuffer = new
Byte[MyHid.Capabilities.OutputReportByteLength];

      if (emulate)
        outputReportBuffer = new Byte[4];

      if (outputReportBuffer.Length >= 4)
      {
        //  Store the report ID in the first byte of the buffer:
        outputReportBuffer[0] = 0; // report ID
        outputReportBuffer[1] = Convert.ToByte(reportType.Write);
        outputReportBuffer[2] = lowerAddr;
        if (MyHid.Capabilities.OutputReportByteLength == 5)
        {

          outputReportBuffer[3] = upperAddr;
```

```
          outputReportBuffer[4] = data;
          dataLoc = 4;
        }
        else
        {
          outputReportBuffer[3] = data;
        }
        if (!silent)
          DebugAdd("Sending " + outputReportBuffer[dataLoc].ToString()
+ " to address " + outputReportBuffer[2]);
        success = SendOutputReport(ref outputReportBuffer);
        if (!silent)
          DebugLine(" succeeded: " + success);
      }

      return success;
    }

    internal Boolean SimpleRead(byte lowerAddr, byte upperAddr, ref
byte data, Boolean silent)
    {
      Boolean successWrite = false;
      Boolean successRead = false;
      Byte[] outputReportBuffer = new
Byte[MyHid.Capabilities.OutputReportByteLength];
      int dataLoc = 4;

      silent = silent | menuItemSilenceOverride.Checked;
      //if (silent == false)
      //debugText.Focus();  // only give the window focus if we need to
output something

      if (MyHid.Capabilities.OutputReportByteLength > 3)
      {
        outputReportBuffer[0] = 0; // reportID
        outputReportBuffer[1] = Convert.ToByte(reportType.Read);
        outputReportBuffer[2] = lowerAddr;

        if (MyHid.Capabilities.OutputReportByteLength == 5)
        {

          outputReportBuffer[3] = upperAddr;
          outputReportBuffer[4] = 0;  // double byte addressing
detected, include high address and then bogus data
        }
        else
        {
          outputReportBuffer[3] = 0;  // single byte addressing
detected, ignore high address
          dataLoc = 3;
        }
      }

      /* So why did we just send a report when we're really reading?
By declaring reportType.Read we're instructing the device
       * to store and enable the address which will then be used to
populate the proper data onto the cypress input pins.
```

```
       * This data will eventually be present in the output report
along with a ValidByte incrementally higher than the last time
       * we issued a read request.  The result may not be immediately
available, so poll the device a few times.  Run this in a
       * seperate process or ensure a timeout to prevent deadlock
       */

      successWrite = SendOutputReport(ref outputReportBuffer);
      if (successWrite == true)
      {
        if (!silent)
          DebugLine("Read address written.  Waiting for result");
        validByte++;
        int retry = 0;
        while (retry < 128)
        {
          ReadInputReport(ref dummyReport);
          if (dummyReport[1] == validByte)
          {
            if (!silent)
              DebugLine("Read success: " + dummyReport[dataLoc]);
            data = dummyReport[dataLoc];
            successRead = true;
            retry = 128;  // kludge to break early
          }
          else
          {
            //DebugLine("Saw unexpected byte: " + dummyReport[1] + ".
Expected: " + validByte);
            Thread.Sleep(10);
            retry++;
          }
        }

      }

      return successRead;
    }

    private Boolean FindTheHid()
    {
      Boolean deviceFound = false;
      String[] devicePathName = new String[128];
      String functionName = "";
      Guid hidGuid = Guid.Empty;
      Int32 memberIndex = 0;
      Int16 myVendorID = Convert.ToInt16("4242", 16);
      Int16 myProductID = Convert.ToInt16("4201", 16);
      Boolean success = false;

      //debugText.Focus();

      try
      {
        Debug.WriteLine("Attempting to open HID Devices");
        myDeviceDetected = false;
```

```
        //  ***
        //  API function: 'HidD_GetHidGuid

        //  Purpose: Retrieves the interface class GUID for the HID
class.

        //  Accepts: 'A System.Guid object for storing the GUID.
        //  ***

        Hid.HidD_GetHidGuid(ref hidGuid);

        functionName = "GetHidGuid";
        Debug.WriteLine(MyDebugging.ResultOfAPICall(functionName));
        Debug.WriteLine("  GUID for system HIDs: " +
hidGuid.ToString());

        //  Fill an array with the device path names of all attached
HIDs.

        deviceFound = MyDeviceManagement.FindDeviceFromGuid(hidGuid,
ref devicePathName);

        //  If there is at least one HID, attempt to read the Vendor ID
and Product ID
        //  of each device until there is a match or all devices have
been examined.

        if (deviceFound)
        {
          memberIndex = 0;

          do
          {
            //  ***
            //  API function:
            //  CreateFile

            //  Purpose:
            //  Retrieves a handle to a device.

            //  Accepts:
            //  A device path name returned by
SetupDiGetDeviceInterfaceDetail
            //  The type of access requested (read/write).
            //  FILE_SHARE attributes to allow other processes to
access the device while this handle is open.
            //  A Security structure or IntPtr.Zero.
            //  A creation disposition value. Use OPEN_EXISTING for
devices.
            //  Flags and attributes for files. Not used for devices.
            //  Handle to a template file. Not used.

            //  Returns: a handle without read or write access.
            //  This enables obtaining information about all HIDs, even
system
            //  keyboards and mice.
            //  Separate handles are used for reading and writing.
```

```
            //  ***

            hidHandle = FileIO.CreateFile(devicePathName[memberIndex],
0, FileIO.FILE_SHARE_READ | FileIO.FILE_SHARE_WRITE, IntPtr.Zero,
FileIO.OPEN_EXISTING, 0, 0);

            functionName = "CreateFile";
            Debug.WriteLine(MyDebugging.ResultOfAPICall(functionName));
            Debug.WriteLine("  Returned handle: " +
hidHandle.ToString());

            if (!hidHandle.IsInvalid)
            {
              //  The returned handle is valid,
              //  so find out if this is the device we're looking for.

              //  Set the Size property of DeviceAttributes to the
number of bytes in the structure.

              MyHid.DeviceAttributes.Size =
Marshal.SizeOf(MyHid.DeviceAttributes);

              //  ***
              //  API function:
              //  HidD_GetAttributes

              //  Purpose:
              //  Retrieves a HIDD_ATTRIBUTES structure containing the
Vendor ID,
              //  Product ID, and Product Version Number for a device.

              //  Accepts:
              //  A handle returned by CreateFile.
              //  A pointer to receive a HIDD_ATTRIBUTES structure.

              //  Returns:
              //  True on success, False on failure.
              //  ***

              success = Hid.HidD_GetAttributes(hidHandle, ref
MyHid.DeviceAttributes);

              if (success)
              {
                Debug.WriteLine("  HIDD_ATTRIBUTES structure filled
without error.");
                Debug.WriteLine("  Structure size: " +
MyHid.DeviceAttributes.Size);
                Debug.WriteLine("  Vendor ID: " +
Convert.ToString(MyHid.DeviceAttributes.VendorID, 16));
                Debug.WriteLine("  Product ID: " +
Convert.ToString(MyHid.DeviceAttributes.ProductID, 16));
                Debug.WriteLine("  Version Number: " +
Convert.ToString(MyHid.DeviceAttributes.VersionNumber, 16));

                //  Find out if the device matches the one we're
looking for.
```

```
                Debug.WriteLine("Looking for: " + myVendorID + " and "
+ myProductID);
                Debug.WriteLine("Looking at " +
MyHid.DeviceAttributes.VendorID + " and " +
MyHid.DeviceAttributes.ProductID);

                if ((MyHid.DeviceAttributes.VendorID == myVendorID) &&
(MyHid.DeviceAttributes.ProductID == myProductID))
                {

                  Debug.WriteLine("  My device detected");

                  //  Display the information in form's list box.

                  myDeviceDetected = true;

                  //  Save the DevicePathName for OnDeviceChange().

                  myDevicePathName = devicePathName[memberIndex];
                }
                else
                {
                  //  It's not a match, so close the handle.

                  myDeviceDetected = false;
                  hidHandle.Close();
                }
              }
              else
              {
                //  There was a problem in retrieving the information.

                Debug.WriteLine("  Error in filling HIDD_ATTRIBUTES
structure.");
                myDeviceDetected = false;
                hidHandle.Close();
              }
            }

            //  Keep looking until we find the device or there are no
devices left to examine.

            memberIndex = memberIndex + 1;
          }
          while (!((myDeviceDetected || (memberIndex ==
devicePathName.Length))));
        }

        if (myDeviceDetected)
        {
          //  The device was detected.
          //  Register to receive notifications if the device is
removed or attached.
```

```
          success =
MyDeviceManagement.RegisterForDeviceNotifications(myDevicePathName,
this.Handle, hidGuid, ref deviceNotificationHandle);

          Debug.WriteLine("RegisterForDeviceNotifications = " +
success);

          //timer1.Enabled = true;

          //  Learn the capabilities of the device.

          MyHid.Capabilities = MyHid.GetDeviceCapabilities(hidHandle);

          if (success)
          {
            //  Find out if the device is a system mouse or keyboard.

            hidUsage = MyHid.GetHidUsage(MyHid.Capabilities);

            //  Get handles to use in requesting Input and Output
reports.

            readHandle = FileIO.CreateFile(myDevicePathName,
FileIO.GENERIC_READ, FileIO.FILE_SHARE_READ | FileIO.FILE_SHARE_WRITE,
IntPtr.Zero, FileIO.OPEN_EXISTING, FileIO.FILE_FLAG_OVERLAPPED, 0);

            functionName = "CreateFile, ReadHandle";
            Debug.WriteLine(MyDebugging.ResultOfAPICall(functionName));
            Debug.WriteLine("  Returned handle: " +
readHandle.ToString());

            if (readHandle.IsInvalid)
            {
              exclusiveAccess = true;
            }
            else
            {
              writeHandle = FileIO.CreateFile(myDevicePathName,
FileIO.GENERIC_WRITE, FileIO.FILE_SHARE_READ | FileIO.FILE_SHARE_WRITE,
IntPtr.Zero, FileIO.OPEN_EXISTING, 0, 0);

              functionName = "CreateFile, WriteHandle";

Debug.WriteLine(MyDebugging.ResultOfAPICall(functionName));
              Debug.WriteLine("  Returned handle: " +
writeHandle.ToString());

              //  Flush any waiting reports in the input buffer.
(optional)

              MyHid.FlushQueue(readHandle);
            }
          }
        }
        else
        {
          //  The device wasn't detected.
```

```
            Debug.WriteLine(" Device not found.");
            timer1.Enabled = false;
          }

          if (myDeviceDetected == true)
            toolStripStatusConnected.Text = "Device connected";
          else
            toolStripStatusConnected.Text = "Device not connected";
          return myDeviceDetected;
        }
        catch (Exception ex)
        {
          throw;
        }
      }

      internal void OnDeviceChange(Message m)
      {
        Debug.WriteLine("WM_DEVICECHANGE");

        try
        {
          if ((m.WParam.ToInt32() == DeviceManagement.DBT_DEVICEARRIVAL))
          {

            //  If WParam contains DBT_DEVICEARRIVAL, a device has been
attached.

            DebugLine("A device has been attached.");

            //  Find out if it's the device we're communicating with.

            if (MyDeviceManagement.DeviceNameMatch(m, myDevicePathName))
            {
              DebugLine("My device attached.");
              toolStripStatusConnected.Text = "Device reattached";  // we
see it, but the handle may have changed
              //rerun findTheHID on next transaction to reenable the
connection
              timer1.Enabled = true;
            }

          }
          else if ((m.WParam.ToInt32() ==
DeviceManagement.DBT_DEVICEREMOVECOMPLETE))
          {

            //  If WParam contains DBT_DEVICEREMOVAL, a device has been
removed.

            DebugLine("A device has been removed.");

            //  Find out if it's the device we're communicating with.

            if (MyDeviceManagement.DeviceNameMatch(m, myDevicePathName))
            {
```

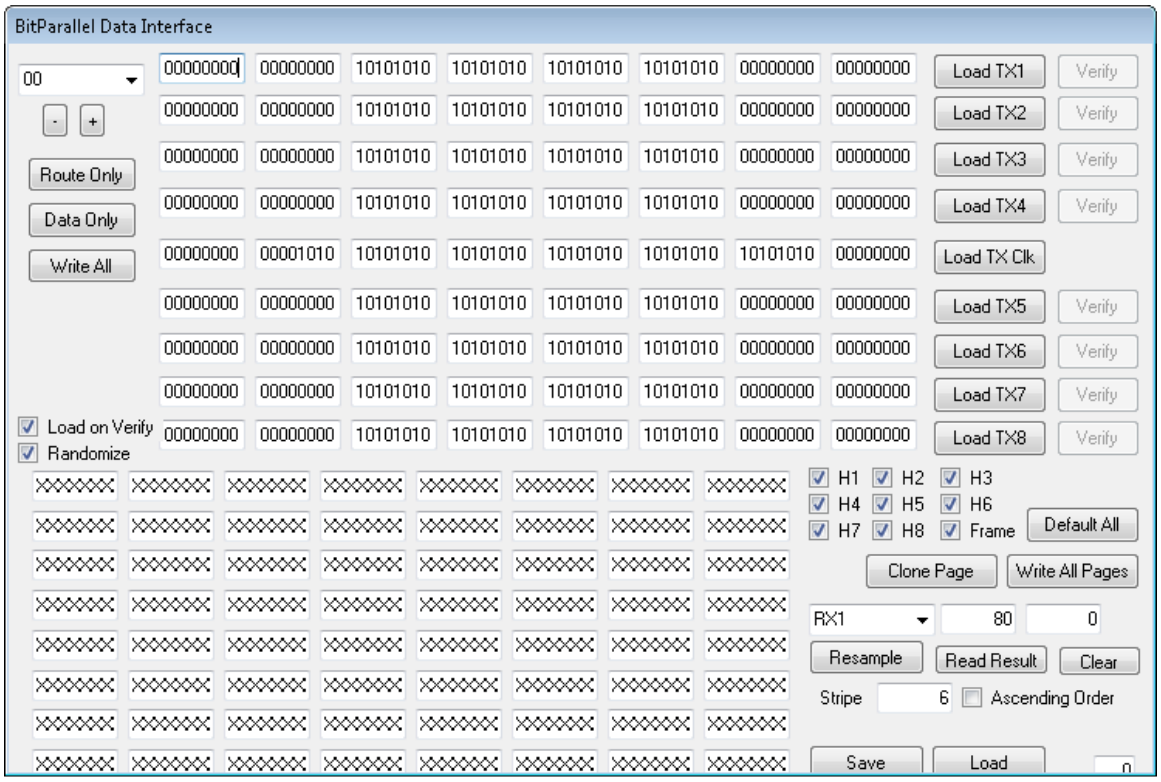
```
            DebugLine("My device removed.");

            //  Set MyDeviceDetected False so on the next data-transfer
attempt,
            //  FindTheHid() will be called to look for the device
            //  and get a new handle.

            myDeviceDetected = false;
            //timer1.Enabled = false;
            toolStripStatusConnected.Text = "Device not connected";
          }
        }
      }
      catch (Exception ex)
      {
        throw;
      }
    }

    private void dataWindowToolStripMenuItem_Click(object sender,
EventArgs e)
    {
      if (frameData.Visible == true)
        frameData.Hide();
      else
      {
        Point loc = this.DesktopLocation;
        Size frameSize = this.Size;
        int x = loc.X + frameSize.Width;
        int y = loc.Y;
        frameData.SetDesktopLocation(x, y);
        frameData.Show();
      }
    }

    private void delayWindowToolStripMenuItem_Click(object sender,
EventArgs e)
    {
      if (frameDelay.Visible == true)
        frameDelay.Hide();
      else
      {
        Point loc = this.DesktopLocation;
        Size frameSize = this.Size;
        int x = loc.X;
        int y = loc.Y + frameSize.Height;
        frameDelay.SetDesktopLocation(x, y);
        frameDelay.Show();
      }
    }

    private void memViewToolStripMenuItem_Click(object sender,
EventArgs e)
    {
      if (frameMemory.Visible == true)
        frameMemory.Hide();
```

```
      else
      {
        frameMemory.Show();
      }
    }

    protected override void WndProc(ref Message m)
    {
      try
      {
        //  The OnDeviceChange routine processes WM_DEVICECHANGE
messages.

        if (m.Msg == DeviceManagement.WM_DEVICECHANGE)
        {
          OnDeviceChange(m);
        }

        //  Let the base form process the message.

        base.WndProc(ref m);
      }
      catch (Exception ex)
      {
        DisplayException(this.Name, ex);
        throw;
      }
    }

    internal static void DisplayException(String moduleName, Exception
e)
    {
      String message = null;
      String caption = null;

      //  Create an error message.

      message = "Exception: " + e.Message + "\r\n" + "Module: " +
moduleName + "\r\n" + "Method: " + e.TargetSite.Name;

      caption = "Unexpected Exception";

      MessageBox.Show(message, caption, MessageBoxButtons.OK);
      Debug.Write(message);
    }

    internal void DebugLine(String newText)
    {
      debugText.AppendText(newText + "\r\n");
    }

    internal void DebugAdd(String newText)
    {
      debugText.AppendText(newText);
    }

    private void timer1_Tick(object sender, EventArgs e)
```

```
    {
      Boolean successRead = false;

      Byte[] inputReportBuffer = new
Byte[MyHid.Capabilities.OutputReportByteLength];
      successRead = ReadInputReport(ref inputReportBuffer);

      if (myDeviceDetected == false)
        return;

      if (successRead == true)
      {
        for (int i = 0; i < inputReportBuffer.Length; i++)
        {
          switch (i)
          {
            case (0): // fallthrough, don't care about the report ID
            case (1): labelValid.Text =
inputReportBuffer[i].ToString();
              break;
            case (2): labelAddrLow.Text =
inputReportBuffer[i].ToString();
              break;
            case (3): if (inputReportBuffer.Length > 4)
              {  // if the report length is 5, we're double byte
addressing
                labelAddrHigh.Text = inputReportBuffer[i].ToString();
                labelData.Text = inputReportBuffer[i + 1].ToString();
              }
              else
              {  // otherwise it's just single byte addressing and the
high address is irrelevant
                labelAddrHigh.Text = "N/A";
                labelData.Text = inputReportBuffer[i].ToString();
              }
              break;
          }
        }
      }
      else
      {
        DebugLine("Read operation appears to have failed.");
      }
    }

    private void synchronizeToolStripMenuItem_Click(object sender,
EventArgs e)
    {
      syncValidByte();
    }

    public void syncValidByte()
    {
      if (myDeviceDetected == true)
      {
        Boolean successRead = false;
        successRead = ReadInputReport(ref dummyReport);
```

```
        if (successRead == true)
        {
          validByte = dummyReport[1];
          DebugLine("Valid Byte reset to: " + validByte);
          labelValid.Text = validByte.ToString();
        }
      }
    }

    private void BitParallel_Move(object sender, EventArgs e)
    {
      Point loc = this.DesktopLocation;
      Size frameSize = this.Size;

      if ((frameDelay.Visible == true) && (delayWindowLock.Checked ==
true))
      {
        int x = loc.X;
        int y = loc.Y + frameSize.Height; ;
        frameDelay.SetDesktopLocation(x, y);
        //DebugLine("Delay's dimensions: " + frameDelay.Width + " by "
+ frameDelay.Height);
        frameDelay.Show();
      }

      if ((frameData.Visible == true) && (dataWindowLock.Checked ==
true))
      {
        int x = loc.X + frameSize.Width;
        int y = loc.Y;
        frameData.SetDesktopLocation(x, y);
        frameData.Show();
      }


    }

    private void
mainWindowTickTimerToolStripMenuItem_CheckStateChanged(object sender,
EventArgs e)
    {
      if (mainWindowTickTimerToolStripMenuItem.Checked == true)
        timer1.Enabled = true;
      else
        timer1.Enabled = false;
    }

    private void debugText_TextChanged(object sender, EventArgs e)
    {
      SuspendLayout();
      Point pt =
debugText.GetPositionFromCharIndex(debugText.SelectionStart);
      if (pt.Y > debugText.Height)
      {
        debugText.ScrollToCaret();
      }
      ResumeLayout(true);
```

```
        }

    }
}
```



The first of two Data Vortex testing windows shown in this work, this data interface allows for the manipulation of data used for transmission and the analysis of captured data. Transmitted data is displayed in this window on a per-packet basis. At the top left is a pull-down selection that allows for the traversal of up-to 16 packets. Data is shown one row per channel, dividied into 8 bytes that comprise the packet, as a literal sequence of 1's and 0's as desired in the transmitted waveform. Routing or header bits, as they are interchangeable called in some components of this work, and the frame can be enabled or disabled per-packet using the check boxes to the center-right of the window. When data is written from the software to the FPGA, the values and destination addresses are calculated automatically based off the system memory map.

Received values can be displayed for a single channel at a time in the grid at the lower left of the window. With the appropriate channel selected, byte offset, stripe value, and ascending selected/unselected, a representation of the data as stored in the receive shift buffer can be displayed in the grid horizontally as a literal interpretation per packet

and vertically as a representation of arrival order. It is not an exact reproduction of arrival as a function of time as slots containing no packets will not generate a frame signal or sample clocks. The values and options used for this region of the software is still very much in the development stage and therefore many of the data values are presented to the user for direct manipulation. As the understanding and confidence of the system improves many of these features can be incorporated directly into the FPGA for intermediate processing or hidden by the software interface.

DataWindow.cs

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using System.Diagnostics;
using System.IO;

namespace BitParallel_HID_Interface
{
  public partial class DataWindow : Form
  {
    private BitParallel bp;

    private Random numGenerator;

    private BPTextArray[] txDataCollection;
    private CheckBox[] routingCollection;
    private BPTextArray[] readCollection;

    private string[, ,] storedData = new string[16, 9, 8];
    private bool[,] storedRouting = new bool[16, 9];
    private int oldPage = 0;

    public DataWindow(BitParallel source)
    {
      InitializeComponent();
      bp = source;
      txDataCollection = new BPTextArray[] { tx1Data, tx2Data, tx3Data,
tx4Data, tx5Data, tx6Data, tx7Data, tx8Data, txclkData };
      routingCollection = new CheckBox[] { checkFrame, checkHeader1,
checkHeader2, checkHeader3, checkHeader4, checkHeader5, checkHeader6,
checkHeader7, checkHeader8 };
```

```
      readCollection = new BPTextArray[] { bpReadArray0, bpReadArray1,
bpReadArray2, bpReadArray3, bpReadArray4, bpReadArray5, bpReadArray6,
bpReadArray7 };
      DefaultAll();
      numGenerator = new Random();
      pageSelect.SelectedIndex = 0;
      rxReadSelect.SelectedIndex = 0;
      textReadAddrHigh.Text = "80";

      for (int i = 0; i < 16; i++)
      {
        storePageValues(i);  // seeding all 16 pages with default data
      }
    }

    #region resetFunctions
    private void DefaultAll()
    {
      foreach (BPTextArray blah in txDataCollection)
      {
        blah.setDefaultData();
      }
      txclkData.setDefaultClk(Convert.ToInt16(textClkOffset.Text));

      foreach (BPTextArray readLoc in readCollection)
      {
        readLoc.setDefaultRead();
      }
      foreach (CheckBox headerBit in routingCollection)
      {
        headerBit.Checked = true;
      }
    }

    private void ClearAll()
    {
      foreach (BPTextArray dataLoc in txDataCollection)
      {
        dataLoc.setClearData();
      }
      foreach (BPTextArray readLoc in readCollection)
      {
        readLoc.setDefaultRead();
      }
      foreach (CheckBox headerBit in routingCollection)
      {
        headerBit.Checked = false;
      }
    }

    private void txButtonDefault_Click(object sender, EventArgs e)
    {
      if (Control.ModifierKeys == Keys.Control)
      {
        ClearAll();
      }
      else
```

```
      {
        DefaultAll();
      }
    }

    private void buttonReadClear_Click(object sender, EventArgs e)
    {
      foreach (BPTextArray row in readCollection)
      {
        row.setDefaultRead();
      }
    }
    #endregion resetFunctions

    private void TXVectorSend(BPTextArray target, byte lowAddr, byte
highAddr)
    {
      byte data = 0;

      for (int i = 0; i < target.textArray.Length; i++)
      {
        data = Convert.ToByte(target.textArray[i].Text, 2);  // double
byte addressing detected, include high address and then data
        bp.SimpleSend(lowAddr, highAddr, data, true);
        lowAddr++;
      }
    }

    private void TXBulkSend()  // transmit all pages
    {
      int oldIndex = pageSelect.SelectedIndex;

      for (int i = 0; i < 16; i++)
      {
        pageSelect.SelectedIndex = i;
        sendFullPageData();
        sendFullPageRouting();
      }

      pageSelect.SelectedIndex = oldIndex;
    }

    private Boolean readVectorByAddress(byte lowAddr, byte highAddr,
int locs, int offset, BPTextArray target, bool ascending)
    {
      Boolean overallSuccess = true;
      for (int i = 0; i < locs; i++)
      {
        byte readResult = 0;
        Boolean success = bp.SimpleRead(lowAddr, highAddr, ref
readResult, true);
        overallSuccess = overallSuccess & success;
        if (success)
        {
          //bp.DebugLine("Data window read back " + readResult);
          String result = Convert.ToString(readResult, 2);
          target.textArray[i + offset].Text = result.PadLeft(8, '0');
```

```
            if (ascending)
              lowAddr++;
            else
              lowAddr--;
          }
          else
            target.textArray[i + offset].Text = "xxxxxxxx";
        }
        return overallSuccess;
      }

      private void TXVectorScramble(BPTextArray target)
      {
        byte data = 0;
        String newValue;

        for (int i = 2; i < 6; i++)
        {
          data = (byte)numGenerator.Next(255);
          newValue = Convert.ToString(data, 2);
          target.textArray[i].Text = newValue.PadLeft(8, '0');
        }
      }

      private void TXButton_Click(object sender, EventArgs e)
      {
        TXButton click_source = (TXButton)sender;
        BPTextArray targetArray = click_source.targetArray;

        int addr = (targetArray.baseMemoryAddress * 2048) +
(pageSelect.SelectedIndex * 8);

        TXVectorSend(targetArray, Convert.ToByte(addr % 256),
Convert.ToByte(addr / 256));
      }

      private void txButtonAll_Click(object sender, EventArgs e)
      {
        sendFullPageData();
      }

      private void sendFullPageData()
      {
        int addr = 0;
        foreach (BPTextArray blah in txDataCollection)
        {
          addr = (blah.baseMemoryAddress * 2048) +
(pageSelect.SelectedIndex * 8);
          TXVectorSend(blah, Convert.ToByte(addr % 256),
Convert.ToByte(addr / 256));
        }

        addr = (txclkData.baseMemoryAddress * 2048) +
(pageSelect.SelectedIndex * 8);
        TXVectorSend(txclkData, Convert.ToByte(addr % 256),
Convert.ToByte(addr / 256));
      }
```

```
    private void txButtonVerify_Click(object sender, EventArgs e)
    {
      TXButton click_source = (TXButton)sender;
      BPTextArray targetArray = click_source.targetArray;

      int readAddr = targetArray.baseReadAddress;
      int writeAddr = targetArray.baseMemoryAddress;

      if (checkRandomize.Checked == true)
      {
        TXVectorScramble(targetArray);
      }

      if (checkVerify.Checked == true)
      {
        TXVectorSend(targetArray, Convert.ToByte(writeAddr % 256),
Convert.ToByte(writeAddr / 256));
      }

      readVectorByAddress(Convert.ToByte(readAddr % 256), 16, 4, 2,
bpReadArray0, true); // upper is hardcoded 16 for RX reading

      for (int i = 2; i < 6; i++)
      {

        if (targetArray.textArray[i].Text ==
bpReadArray0.textArray[i].Text)
          bpReadArray0.textArray[i].BackColor = Color.Green;
        else
          bpReadArray0.textArray[i].BackColor = Color.Red;

        // exception made for the first 2 bits of the first byte
        if ((i == 2) & (bpReadArray0.textArray[i].BackColor ==
Color.Red))
        {
          String readTail =
bpReadArray0.textArray[i].Text.Substring(2);
          String sourceTail =
targetArray.textArray[i].Text.Substring(2);

          // if the remainder of the byte passes, the error's in the
first 2 bits, flag as yellow
          if (readTail.Equals(sourceTail))
          {
            bpReadArray0.textArray[i].BackColor = Color.Yellow;
          }
        }

      }
    }

    private void dataReadButton_Click(object sender, EventArgs e)
    {

      bp.syncValidByte();
```

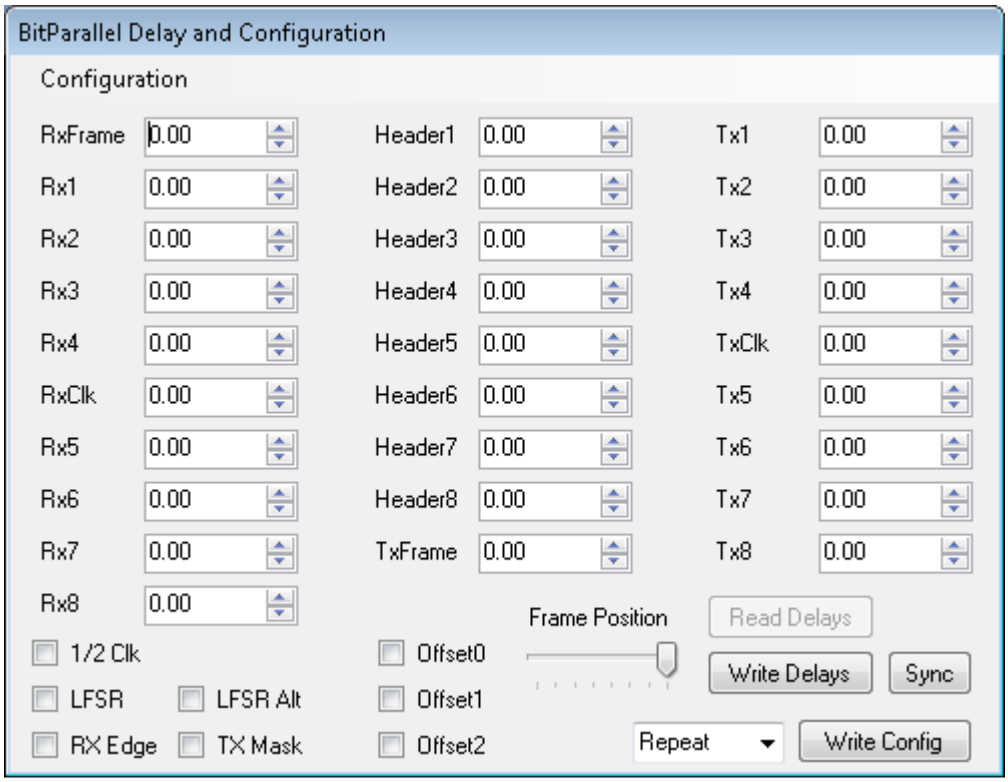
```
      int lowNum = Convert.ToInt16(textReadAddrLow.Text);
      int highNum = Convert.ToInt16(textReadAddrHigh.Text);
      int stripeInt = Convert.ToInt16(textReadStripeCount.Text);
      byte stripeByte = Convert.ToByte(textReadStripeCount.Text);
      byte lowAddr = (byte)(lowNum % 256);
      byte highAddr = (byte)(highNum % 256);
      bool ascending = checkReadOrder.Checked;

      for (int i = 0; i < readCollection.Length; i++)
      {
        readVectorByAddress(lowAddr, highAddr, stripeInt, 0,
readCollection[i], ascending);
        if (ascending)
          lowAddr += stripeByte;
        else
          lowAddr -= stripeByte;
      }
    }

    private void pageSelect_SelectedIndexChanged(object sender,
EventArgs e)
    {
      storePageValues(oldPage);
      oldPage = pageSelect.SelectedIndex;
      loadPageValues(oldPage);
    }

    private void storePageValues(int page)
    {
      int row = 0;  // 9 rows, tx1-8 + clk,
      foreach (BPTextArray blah in txDataCollection)
      {
        for (int i = 0; i < 8; i++)  // 8 bytes per row, store string
data rather than byte values
        {
          storedData[page, row, i] = blah.textArray[i].Text;
        }
        row++;
      }

      for (int i = 0; i < routingCollection.Length; i++)
      {
        storedRouting[page, i] = routingCollection[i].Checked;
      }

    }

    private void loadPageValues(int page)
    {
      int row = 0;
      foreach (BPTextArray blah in txDataCollection)
      {
        for (int i = 0; i < 8; i++)
        {
          blah.textArray[i].Text = storedData[page, row, i];
        }
        row++;
```

```
      }

      for (int i = 0; i < routingCollection.Length; i++)
      {
        routingCollection[i].Checked = storedRouting[page, i];
      }
    }

    private void btnRoute_Click(object sender, EventArgs e)
    {
      sendFullPageRouting();

    }

    private void sendFullPageRouting()
    {
      int page = pageSelect.SelectedIndex;
      int addr = page * 2;
      int data = 0;
      int value = 1;
      for (int i = 0; i < 8; i++)
      {
        if (routingCollection[i].Checked == true)
          data = data + value;

        value = value * 2;
      }

      bp.SimpleSend(Convert.ToByte(addr), Convert.ToByte(1),
Convert.ToByte(data), true);
      addr++;
      if (routingCollection[8].Checked == true)
        bp.SimpleSend(Convert.ToByte(addr), Convert.ToByte(1),
Convert.ToByte(1), true);
      else
        bp.SimpleSend(Convert.ToByte(addr), Convert.ToByte(1),
Convert.ToByte(0), true);
    }

    private void btnResample_Click(object sender, EventArgs e)
    {
      bp.SimpleSend((byte)0, (byte)0, (byte)4, true);
    }

    private void btnCloneAll_Click(object sender, EventArgs e)
    {
      for (int i = 0; i < 16; i++)
      {
        storePageValues(i);  // seeding all 16 pages with default data
      }
    }

    private void btnWriteAllPages_Click(object sender, EventArgs e)
    {
      TXBulkSend();
    }
```

```
    private void btnWriteAllPage_Click(object sender, EventArgs e)
    {
      sendFullPageRouting();
      sendFullPageData();
    }

    private void rxReadSelect_SelectedIndexChanged(object sender,
EventArgs e)
    {
      int readAddr = 80 + rxReadSelect.SelectedIndex;
      textReadAddrHigh.Text = readAddr.ToString();
    }

    private void WriteDataSet(string filename)
    {
      TextWriter tw = new StreamWriter(filename);
      for (int page = 0; page < 16; page++)
      {
        int row = 0;  // 9 rows, tx1-8 + clk
        foreach (BPTextArray dataRow in txDataCollection)
        {
          for (int i = 0; i < 8; i++)  // 8 bytes per row, store string
data rather than byte values
          {
            tw.Write(storedData[page, row, i] + ",");
          }
          row++;
          tw.WriteLine("");
        }

        for (int i = 0; i < routingCollection.Length; i++)
        {
          tw.Write(storedRouting[page, i] + ",");
        }
        tw.WriteLine("");
      }

      tw.Close();
    }

    private void ReadDataSet(string filename)
    {
      StreamReader sr = new StreamReader(filename);

      try
      {
        for (int page = 0; page < 16; page++)
        {
          int row = 0;  // 9 rows, tx1-8 + clk
          foreach (BPTextArray dataRow in txDataCollection)
          {
            string[] readText = sr.ReadLine().Split(',');
            if (readText.Length > 8)
            {
              for (int i = 0; i < 8; i++)  // 8 bytes per row, store
string data rather than byte values
              {
```

```
                storedData[page, row, i] = readText[i];
              }
              row++;
            }
          }
          string[] headerBools = sr.ReadLine().Split(',');
          if (headerBools.Length > 8)
            for (int i = 0; i < routingCollection.Length; i++)
            {
              try
              {
                storedRouting[page, i] = Boolean.Parse(headerBools[i]);
              }
              catch { }

            }
        }
      }
      catch { }

      sr.Close();
    }

    private void btnSave_Click(object sender, EventArgs e)
    {
      SaveFileDialog sfd = new SaveFileDialog();
      string filename = "StoredData.txt";
      sfd.FileName = filename;
      sfd.Filter = "Text Files|*.txt|All Files|*.*";

      if (sfd.ShowDialog() != DialogResult.Cancel)
      {
        filename = sfd.FileName;
      }

      storePageValues(pageSelect.SelectedIndex);  //write current ui
values to the backend storage set
      WriteDataSet(filename);
    }

    private void btnLoad_Click(object sender, EventArgs e)
    {
      OpenFileDialog ofd = new OpenFileDialog();
      string filename = "StoredData.txt";
      ofd.FileName = filename;
      ofd.Filter = "Text Files|*.txt|All Files|*.*";

      if (ofd.ShowDialog() != DialogResult.Cancel)
      {
        filename = ofd.FileName;
      }
      ReadDataSet(filename);

      loadPageValues(pageSelect.SelectedIndex);  //refresh ui page from
the loaded data set
    }
```

```
    private void btnPageChange_Click(object sender, EventArgs e)
    {
      int currentPage = pageSelect.SelectedIndex;
      if (sender.Equals(btnPageDown))
      {
        pageSelect.SelectedIndex = (currentPage + 15) % 16;  // c#
modulo operation does not flip negatives back to positive, grr
      }
      else
      {
        pageSelect.SelectedIndex = (currentPage + 1) % 16;
      }

    }


  }
}
```



This is the second of the core Data Vortex testing windows used for the assignment of delay values to the respective channels and the setting of many configuration options. The number fields represent the amount of delay required for the respective channel, measured in nanoseconds. The fields are automatically locked to only present the range of values supported by the system. The button 'Write Delays' pushes all of these values to the proper memory locations and initiates the control operation that will iterate over those memory addresses and push the values to the appropriate delay chips. Single on/off bit options can be selected with the checkboxes or more complicated features, such as the Frame Position slider, can be encoded into memory values which are copied into the configuration register COMMAND_STATE.

DelayWindow.cs

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
```

```
using System.Linq;
using System.Text;
using System.Windows.Forms;
using System.IO;

namespace BitParallel_HID_Interface
{
  public partial class DelayWindow : Form
  {
    private BitParallel bp;
    private BPNumericUpDown[] bpDelay;


    public DelayWindow(BitParallel source)
    {
      InitializeComponent();
      bp = source;

      bpDelay = new BPNumericUpDown[]{bpDelay01, bpDelay02, bpDelay03,
bpDelay04, bpDelay05, bpDelay06,
                                                bpDelay07, bpDelay08,
bpDelay09, bpDelay10, bpDelay11, bpDelay12,
                                                bpDelay13, bpDelay14,
bpDelay15, bpDelay16, bpDelay17, bpDelay18,
                                                bpDelay19, bpDelay20,
bpDelay21,  bpDelay22, bpDelay23, bpDelay24,

bpDelay25,bpDelay26,bpDelay27,bpDelay28};
    }

    private void buttonWrDelays_Click(object sender, EventArgs e)
    {
      /*
       * ToDo:  convert function over to SimpleSend compatibility
       *
       */
      Boolean success = false;

      foreach (BPNumericUpDown blah in bpDelay)
      {
        int addr = blah.targetAddress;
        Byte[] outputReportBuffer = new
Byte[bp.MyHid.Capabilities.OutputReportByteLength];
        int dataLoc = 3;

        if (bp.emulate)
          outputReportBuffer = new Byte[4];

        int delayVal = (int)(blah.Value * 100);

        int lowByte = delayVal % 256;
        int highByte = delayVal / 256;

        bp.DebugLine("Converting " + delayVal + ". Modulo is " +
lowByte + " and divide is " + highByte);

        if (outputReportBuffer.Length >= 4)
```

```
        {

          //  Store the report ID in the first byte of the buffer:
          outputReportBuffer[0] = 0; // report ID
          outputReportBuffer[1] = Convert.ToByte(reportType.Write);
          outputReportBuffer[2] = Convert.ToByte(addr % 256);  // low
byte, if bigger than 256
          if (bp.MyHid.Capabilities.OutputReportByteLength == 5)
          {

            outputReportBuffer[3] = Convert.ToByte(addr / 256);  //
high byte if it exists
            outputReportBuffer[4] = Convert.ToByte(lowByte);  // double
byte addressing detected, include high address and then data
            dataLoc = 4;
          }
          else
          {
            outputReportBuffer[3] = Convert.ToByte(lowByte);  // single
byte addressing detected, ignore high address
          }
          bp.DebugLine("Sending low byte " +
outputReportBuffer[dataLoc].ToString() + " to address " +
outputReportBuffer[2]);
          success = bp.SendOutputReport(ref outputReportBuffer);  //
send the low portion of the delay
          outputReportBuffer[2]++;                                  //
increment to next address
          outputReportBuffer[dataLoc] = Convert.ToByte(highByte);
// send the high portion of the delay
          bp.DebugLine("Sending high byte " +
outputReportBuffer[dataLoc].ToString() + " to address " +
outputReportBuffer[2]);
          success = bp.SendOutputReport(ref outputReportBuffer);
        }
      }
      bp.DebugLine("Output operation: " + success + "\r\n");

      if (success == true)
      {
        bp.SimpleSend(0, 0, 2, true);
      }
    }

    private void buttonSync_Click(object sender, EventArgs e)
    {
      bp.SimpleSend(0, 0, 3, true);
    }

    private void buttonConfig_Click(object sender, EventArgs e)
    {
      byte addr2Byte = 0;
      byte addr3Byte = 0;
      byte addr4Byte = 0;
      byte addr5Byte = 0;

      // Byte #2 contains:
```

```
      if (checkTXMask.Checked == true)
      {   // Bit 0 (this one's kinda weird and tied to the packet
formatter, both of which are buggy right now)
        addr2Byte += 1;
      }
      if (checkOffset0.Checked == true)
      {   // Bit 1
        addr2Byte += 2;
      }
      if (checkOffset1.Checked == true)
      {   // Bit 2
        addr2Byte += 4;
      }
      if (checkOffset2.Checked == true)
      {   // Bit 3
        addr2Byte += 8;
      }

      // Byte #3 contains:
      if (checkLFSR.Checked == true)
      {   // Bit 0.  Enable LFSR
        addr3Byte += 1;
      }
      if (checkLFSRAlt.Checked == true)
      {   // Bit 0.  Enable LFSR
        addr3Byte += 2;
      }
      if (checkHalfClk.Checked == true)
      {   // Bit 0.  Enable LFSR
        addr3Byte += 4;
      }
      if (checkRXEdge.Checked == true)
      {
        addr3Byte += 16;
      }
      // bit 5 is the tx mode enable, bits 6 and 7 are the encoded
frequency, single, 2x, 4x, 16x
      int comboSelection = comboSendMode.SelectedIndex;
      if (comboSelection == 1)
      {
        addr3Byte += 32;
      }
      if (comboSelection == 2)
      {
        addr3Byte += 96;
      }
      if (comboSelection == 3)
      {
        addr3Byte += 160;
      }
      if (comboSelection == 4)
      {
        addr3Byte += 224;
      }

      //// Byte #4 contains:
      //if (checkHeader1.Checked == true)
```

```
      //{
      //    addr4Byte += 1;
      //}
      //if (checkHeader2.Checked == true)
      //{
      //    addr4Byte += 2;
      //}
      //if (checkHeader3.Checked == true)
      //{
      //    addr4Byte += 4;
      //}
      //if (checkHeader4.Checked == true)
      //{
      //    addr4Byte += 8;
      //}
      //if (checkHeader5.Checked == true)
      //{
      //    addr4Byte += 16;
      //}
      //if (checkHeader6.Checked == true)
      //{
      //    addr4Byte += 32;
      //}
      //if (checkHeader7.Checked == true)
      //{
      //    addr4Byte += 64;
      //}
      //if (checkHeader8.Checked == true)
      //{
      //    addr4Byte += 128;
      //}

      // Byte #5 contains:
      addr5Byte = (byte)(trackFramePos.Value); // get slider position,
set as bits 2-0 = Frame Offset

      bp.DebugLine("Setting header bits with value: " + addr3Byte);
      bp.DebugLine("Setting frame offset with value: " + addr2Byte);

      bp.SimpleSend(2, 0, addr2Byte, false);
      bp.SimpleSend(3, 0, addr3Byte, false);
      bp.SimpleSend(4, 0, addr4Byte, false);
      bp.SimpleSend(5, 0, addr5Byte, false);

    }

    private void loadDelaySetToolStripMenuItem_Click(object sender,
EventArgs e)
    {
      OpenFileDialog ofd = new OpenFileDialog();
      string filename = "StoredDelays.txt";
      ofd.FileName = filename;
      ofd.Filter = "Text Files|*.txt|All Files|*.*";

      if (ofd.ShowDialog() != DialogResult.Cancel)
      {
        filename = ofd.FileName;
```

```
      }
      ReadDelaySet(filename);

    }

    private void saveDelaySetToolStripMenuItem_Click(object sender,
EventArgs e)
    {
      SaveFileDialog sfd = new SaveFileDialog();
      string filename = "StoredDelays.txt";
      sfd.FileName = filename;
      sfd.Filter = "Text Files|*.txt|All Files|*.*";

      if (sfd.ShowDialog() != DialogResult.Cancel)
      {
        filename = sfd.FileName;
      }

      WriteDelaySet(filename);
    }

    private void ReadDelaySet(string filename)
    {
      StreamReader sr = new StreamReader(filename);

      try
      {
        foreach (BPNumericUpDown delayNum in bpDelay)
        {
          string readText = sr.ReadLine();
          try
          {
            delayNum.Value = Decimal.Parse(readText);
          }
          catch { }
        }
      }
      catch { }

    }

    private void WriteDelaySet(string filename)
    {
      TextWriter tw = new StreamWriter(filename);
      foreach (BPNumericUpDown delayNum in bpDelay)
      {
        tw.WriteLine(delayNum.Value.ToString());
      }
      tw.Close();
    }


  }
}
```

REFERENCES

[1] D.C. Keezer, C. Gray, A. Majid, N. Taher, “Low-cost multi-gigahertz test systems using CMOS FPGAs and PECL,” pp. 152-157, Proc. of Design, Automation, and Test in Europe, March 2005.

[2] Maxwell, P., Hartano, I., Bentz, L. “Comparing functional and structural tests,” *Proc. International Test Conference*, pp. 400-407, 2000.

[3] A.T. Sivarar, M. Shimanouchi, H. Maassen, R. Jackson, “Tester architecture for the source synchronous bus,” *Proc. of the Intl. Test Conf.*, pp. 738-747, Oct. 2004.

[4] M. Abramovici, M.A. Breuer, A.D. Friedman, *Digital Systems Testing and Testable Design* (revised printing), IEEE Press: New York, 1990.

[5] M. Bushnell, V. Agrawal, Essentials of Electronic Testing for Digital, Memory, and Mixed-Signal VLSI Circuits, Kluwer Academic Press: Boston/Dordrecht/Longon, 2000.

[6] V93000 SOC Pin Scale 800 Digital Card: Technical Specifications, Available: http://www1.verigy.com/cnsmprod/groups/public/@supp/@v93k/documents/file/wcmd_001342.pdf [Accessed: Dec. 15, 2009. Site account required].

[7] D. Goswami, N. Mukherjee, “At-speed scan tests: reality or fantasy?” *Proc. of the Intl. Test Conf.*, Panel 1 overview, Oct. 2007.

[8] Verigy, V93000 SOC: High-Speed I/O Test (product overview), Available: http://www1.verigy.com/ate/products/V93000/pcc/digital/hsio/index.htm [Accessed: Dec. 15, 2009].

[9] Centellax, Inc., Centellax Test & Measurement Product Catalog, Centellax Inc., 2012. Available: http://www.centellax.com/sites/centellax.com/files/2012_brochure_final.pdf

[10] Keezer, D.C.; Minier, D.; Paradis, M.; Binette, L., "Modular extension of ATE to 5 Gbps," *Proc. of the Intl. Test Conf.*, pp. 748-757, 26-28 Oct. 2004.

[11] D.C. Keezer, D. Minier, P. Ducharme, "Source-synchronous testing of multilane PCI Express and HyperTransport buses," *IEEE Design & Test of Computers*, Vol. 23, No. 1, pp. 46-57, Jan-Feb. 2005.

[12] D.C. Keezer, C. Gray, A. Majid, D. Minier, P. Ducharme, “A development platform and electronic modules for automated test up to 20 gbps,” *Proc. of the Intl. Test Conf.*, Paper 14.3 pp.1-11, November 2009.

[13] Molavi, S.; Evans, A.; Clancy, R., "Protocol aware test methodologies using today's ATE," *Asian Test Symposium, 2008. ATS '08*. 17th , vol., no., pp.273-273, 24-27 Nov. 2008.

[14] Evans, A.C., "The new ATE: Protocol aware," *Proc. of the Intl. Test Conf* ., Paper 20.1 pp.1-10, 21-26 Oct. 2007.

[15] Protocol Aware Test System KT-72000 FINN product literature, Konrad Technologies, Available: http://www.konrad-technologies.de/english/test-systems/semiconductor-tester-finn/kt-7200-pat [Retrieved May 22, 2012].

[16] Courbis, A.-L.; Santucci, J.-F.; Giambiasi, N.; , "Automatic behavioral test pattern generation for digital circuits ," *Test Symposium, 1992. (ATS '92), Proceedings., First Asian (Cat. No.TH0458-0)* , vol., no., pp.112-117, 26-27 Nov 1992.

[17] Bierman., H. ; "VLSI test gear keeps pace with chip advances," Electronics, April 19, 1984.

[18] West, B.; Napier, T.; , "Sequencer Per Pin test system architecture," Test Conference, 1990. Proceedings., International , vol., no., pp.355-361, 10-14 Sep 1990.

[19] Miczo, Alexander. *Digital Logic Testing and Simulation*. New York: Harper & Row, 1986.

[20] D. Dempster, M. Stuart; *Verification Methodology Manual – Techniques for Verifying HDL Designs*, Hapshire: Teamwork International and TransEDA Limited, 2002.

[21] Hewlett-Packard, *HP83000 Digital Test System, Model F660 System Description*, Part #5091-2357E, Hewlett-Packard Gmbh, May 1992.

[22] Hewlett-Packard, Testing Digital Devices To The Limits, HP 83000 Digital Test System, Model F660 Technical Data, Hewlett-Packard Gmbh, July 1993.

[23] Hewlett-Packard, HP83000 Digital Test System, Using the HP 83000 Model F660, Hewlett-Packard Gmbh, June 1995.

[24] Shmoo plot, http://en.wikipedia.org/wiki/Shmoo_plot

[25] Moore, G. E.; , "Cramming more components onto integrated circuits," Electronics, Volume 38, Number 8, April 19, 1965.

[26] *Butts, Mike;* "Logic Emulation and Prototyping: It's the Interconnect (Rent rules), Given on August 2010 at RAMP Wrap, August 2010, http://ramp.eecs.berkeley.edu/Publications/RAMP2010_MButts20Aug%20(Slides, %208-25-2010).pptx (retrieved May 21, 2012).

[27] Jahangiri, J.; Mukherjee, N.; Wu-Tung Cheng; Mahadevan, S.; Press, R.; , "Achieving High Test Quality with Reduced Pin Count Testing," *Test Symposium, 2005. Proceedings. 14th Asian* , pp. 312- 317, 18-21 Dec. 2005.

[28] IEEE Standard Test Access Port and Boundary-Scan Architecture-Description, IEEE Std 1149.1-2001, 2001.

[29] XJTAG Ltd., "JTAG – A Technical Overview," 2012, Available: http://www.xjtag.com/support-jtag/jtag-technical-guide.php [Retrieved May 6, 2012].

[30] Bleeker, Harry, and Peter van den Eijnden. *Boundary-scan Test: a Practical Approach*. Dordrecht: Kluwer, 1993.

[31] JTAG image, http://en.wikipedia.org/wiki/Joint_Test_Action_Group

[32] Agrawal, V.D.; Kime, C.R.; Saluja, K.K.; "A tutorial on built-in self-test. I. Principles," *Design & Test of Computers, IEEE* , vol.10, no.1, pp.73-82, Mar 1993.

[33] Agrawal, V.D.; Kime, C.R.; Saluja, K.K.; , "A tutorial on built-in self-test. 2. Applications," *Design & Test of Computers, IEEE* , vol.10, no.2, pp.69-77, June 1993.

[34] Stroud, C. E., *A Designer's Guide to Built-in Self-test*, Kluwer Academic Publishers, 2002.

[35] Virtex-5 FPGA User Guide, UG190 v5.4, Xilinx, Inc., San Jose, CA, March 16, 2012.

[36] W. Marx, V. Aggarwal, "FPGAs are everywhere – in design, test & control," *RTC Magazine*, April 2008. (Also available: http://zone.ni.com/devzone/cda/pub/p/id/401).

[37] Altera, "Semiconductor Automated Test Equipment," Available: http://www.altera.com/end-markets/test-measurements/test-products/ate/tes-ate.html#figure1 [Accessed Dec. 15, 2009].

[38] J.S. Davis, D.C. Keezer, "Multi-Purpose Digital Test Core Utilizing Programmable Logic," *Proc. of the Intl. Test Conf.*, pp. 438-445, October 2002.

[39] Keezer, D.; Gray, C.; Majid, A.; Taher, N.; , "Implementing multi-gigahertz test systems using CMOS FPGAs and PECL components," *Solid-State Circuits Conference, 2005. ESSCIRC 2005. Proceedings of the 31st European* , vol., no., pp. 291- 294, 12-16 Sept. 2005.

[40] Virtex-5 Family Overview, DS100, v5.0, Xilinx, Inc., San Jose, CA, February 6, 2009.

[41] D. C. Keezer, Q. Zhou, "Test Support Processors for Enhanced Testability of High-Performance Circuits," *Proc. of the Intl. Test Conf.*, pp. 801-809, October 1999.

[42] A.M. Majid, D.C. Keezer, "An Improved Low-Cost 6.4 Gbps Wafer-Level Tester" *Proc. of the 6th IEEE Electronics Packaging Technology Conference* (EPTC), pp.814-819, December 2005.

[43] D. C. Keezer, D. Minier, M.C. Caron, "Multiplexing ATE Channels for Production Testing at 2.5 Gbps," *IEEE Design & Test of Computers,* Vol. 21, No. 4, pp. 288-301, July-Aug. 2004.

[44] Aggarwal, V.; , "Protocol Aware ATE with FPGA-based hardware," *AUTOTESTCON, 2008 IEEE* , vol., no., pp.595-597, 8-11 Sept. 2008.

[45] Aggarwal, V., "Protocol aware ATE with FPGA-based hardware," *AUTOTESTCON, 2008 IEEE,* pp.595-597, 8-11 Sept. 2008.

[46] A.Evans, "The New ATE: Protocol Aware," *Proc. of the Intl. Test Conf.,* Panel 5.1, Oct. 2007.

[47] S. Sunter, "Protocol-Aware ATE: Complement or Competitor for Structural Testing?" *Proc. of the Intl. Test Conf.*, Panel 5.2, Oct. 2007.

[48] Burlison, Crouch, Ritchie, “Protocol Aware Test.. It Has a Role, But Where? And How?” *Proc. of the Intl. Test Conf.*, Panel 5.3, Oct. 2007.

[49] G. Conner, “Is a Protocol Aware Test System Feasible?” *Proc. of the Intl. Test Conf.*, Panel 5.4, Oct. 2007.

[50] J. Rivoir, “Protocol-Aware ATE Enables Cooperative Test between DUT and ATE for Improved TTM and Test Quality,” *Proc. of the Intl. Test Conf.*, Panel 5.5, pp. 1-2, Oct. 2007.

[51] T.M. Mak, Mike Tripp, Anne Meixner, “Testing gbps interfaces without a gigahertz tester,” pp. 278-286, *IEEE Design & Test of Computers*, July-August 2004.

[52] K. Mohanram, N.A. Touba, “Eliminating non-determinism during test of high-speed source synchronous differential buses,” *Proc. of the 21st IEEE Test Symposium*, pp. 121-127, April-May 2004.

[53] Ravi Budruk, Don Anderson, Tom Shanley, *PCI Express System Architecture*, MindShare Inc., 2006.

[54] A.Li, J. Faucher, D.V. Plant, "Burst-mode clock and data recovery in optical multiaccess networks using broad-band PLLs," *IEEE Photonics Technology Letters*, Vol. 18, NO. 1, pp73-75, Jan. 1, 2006.

[55] J. Faucher, M. Mony, D.V. Plant, “Test setup for optical burst-mode receivers,” *Proc. IEEE Lightwave Technologies in Instrumentation and Measurement Conf.*, 2004, pp. 123-128.

[56] Jay Trodden, Don Anderson, *HyperTransport System Architecture*, MindShare, Inc. 2003.

[57] MC10EP445, MC100EP445 3.3V/5V ECL 8-Bit Serial/Parallel Converter, MC10EP445/D Rev. 14, Semiconductor Components Industries, LLC, February, 2011.

[58] Williams, T.W.; Parker, K.P.; , "Design for testability—A survey," Proceedings of the IEEE , vol.71, no.1, pp. 98- 112, Jan. 1983.

[59] B.A. Small, O. Liboiron-Ladouceur, A. Shacham, J.P. Mack, K. Bergman, "Demonstration of a Complete 12-Port Terabit Capacity Optical Packet Switching Fabric," OFC 2005 OWK1 (Mar 2005).

[60] A.Shacham, B.A. Small, O. Liboiron-Ladouceur, K. Bergman, "A Fully Implemented 12x12 Data Vortex Optical Packet Switching Interconnection Network," *J. Lightwave Technol*. 23 (10) 3066-3075 (Oct 2005).

[61] Lu, W.; Liboiron-Ladouceur, O.; Small, B.A.; Bergman, K., "Cascading switching nodes in data vortex optical packet interconnection network," *Electronics Letters* , vol.40, no.14, pp. 895-897, 8 July 2004.

[62] O. Liboiron-Ladouceur, C. Gray, D.C. Keezer, and K. Bergman, "Bit-Parallel Message Exchange and Data Recovery in Optical Packet Switched Interconnection Networks," *IEEE Photonics Technology Letters*, Vol. 18, No. 6, pp 779-881, Mar. 2006.

[63] Gray, C.E.; Liboiron-Ladouceur, O.; Keezer, D.C.; Bergman, K., "Co-development of test electronics and PCI Express interface for a multi-Gbps optical switching network," *Proc. of the Intl. Test Conf.*, Paper 22.1, pp. 1-9, 21-26 Oct. 2007.

[64] Amphenol RF product literature, SMP Connector Series, Available: http://www.amphenolrf.com/products/smp.asp?N=0&sid=4FBAD70034E6617F&.

[65] Johnson, H.; Graham, M.; , "High-Speed Digital Design: A Handbook of Black Magic," Prentice Hall PTR, New Jersey, 1993.

[66] Virtex-6 FPGA PCB Design Guide, UG373 v1.2, Xilinx, Inc., San Jose, CA, June 10, 2010.

[67] Rogers Corporation, "RO4000 Series High Frequency Circuit Materials," Publication #92-004 Rev2, November 2011.

[68] High-Speed Serial I/O Made Simple, Xilinx, http://www.xilinx.com/publications/archives/books/serialio.pdf (retrieved May 18, 2012).

[69] PXPIPE White Paper, AN10372 Rev 1.0, Koninklijke Philips Electronics N.V., 30 April 2005.

[70] Gray, C.E.; Liboiron-Ladouceur, O.; Keezer, D.C.; Bergman, K., "Test electronics for a multi-gbps optical packet switching network," *Electronics Packaging Technology Conference, 2006. EPTC '06. 8th* , pp.373-378, 6-8 Dec. 2006.

[71] Wang, H.; Bergman, K.; Gray, C.; Keezer, D.C.; , "*Demonstration of end-to-end bit-parallel memory transactions across the ultra-low latency data vortex optical packet switch,*" Optical Fiber Communication (OFC), collocated National Fiber Optic Engineers Conference, 2010 Conference on (OFC/NFOEC) , vol., no., pp.1-3, 21-25 March 2010.

[72] M.C. Cardakli and A. E. Willner, "Synchronization of a network element for optical packet switching using optical correlators and wavelength shifting," IEEE Photon. Technol. Letters., vol. 14, pp. 1375-1378, 2002.

[73] L. A. Bergman, C. Yeh, J. Morookian, "Advances in multichannel multiGbytes/s bit-parallel WDM single fiber link," *IEEE Trans. Adv. Packag.*, vol. 24, no. 4, pp. 456-462, Nov. 2001.

[74] A. P. Togneri, M. E. Vieira Segatto, "All optical bit parallel transmission systems," *Proc. SMBO/IEEE MTT-S Int.*, vol. 1, Sep. 2003, pp. 367-372.

[75] M.R. Ahmadi, A. Amirkhany, R. Harjani, "A 5Gbps 0.14um CMOS Pilot-Based Clock and Data Recovery Scheme for High-Speed Links," Solid-State Circuits, IEEE Journal of, vol. 45, no. 8, pp. 1533-1541, Aug. 2010

[76] B.J. Shastri, D.V. Plant, "5/10-Gb/s Burst-Mode Clock and Data Recovery Based on Semiblind Oversampling for PONSs: Theoretical and Experimental," Selected Topics in Quantum Electronics, IEEE Journal of, vol. 16, no. 5, pp. 1298-1320, Sept.-Oct. 2010

[77] A. Shacham, K. Bergman, "An Experimental Validation of a Wavelength-Striped, Packet Switched, Optical Interconnection Network," *Lightwave Technology, Journal of*, vol.27, no.7, pp.841-850, April1, 2009.

[78] R. Hemenway, R. Grzybowski, C. Minkenberg, R. Luijten, "Optical-packet-switched interconnect for supercomputer applications," J. Opt. Netw. 3, 900-913 (2004).

[79] M. Nakamura, Y. Imai, Y. Umeda, Jun Endo, Y. Akatsu, "1.25-Gb/s burst-mode receiver ICs with quick response for PON systems," Solid-State Circuits, IEEE Journal of , vol.40, no.12, pp. 2680- 2688, Dec. 2005.

[80] Wei-Zen Chen, Ruei-Ming Gan, Shih-Hao Huang, "A Single-Chip 2.5-Gb/s CMOS Burst-Mode Optical Receiver," Circuits and Systems I: Regular Papers, IEEE Transactions on , vol.56, no.10, pp.2325-2331, Oct. 2009.

[81] Gaowei Gu, En Zhu, Ye Lin, "A Gated VCO for 10Gb/s PON Systems in 0.18μm CMOS," Future Networks, 2010. ICFN '10. Second International Conference on, vol., no., pp.261-264, 22-24 Jan. 2010.

[82] H. Furukawa, N. Wada, H. Fujinuma, H. Iiduka, T. Miyazaki, "Instantaneous-locking 8-channel arrayed 10 Gbps burst- mode optical packet receiver and 80 (8λ x 10) Gbps wide-colored optical packet transmitter," Lasers and Electro-Optics, 2007. CLEO 2007. Conference on, vol., no., pp.1-2, 6-11 May 2007.

[83] Hyde, John, USB Design-by-example: a practical guide to building IO devices, John Wiley & Sons, New York, 1999.

PUBLICATIONS

Keezer, D.C.; Gray, C.; Majid, A.; Taher, N.; , "Low-cost multi-gigahertz test systems using CMOS FPGAs and PECL," *Design, Automation and Test in Europe, 2005. Proceedings* , vol., no., pp. 152- 157 Vol. 1, 7-11 March 2005

Keezer, D.; Gray, C.; Majid, A.; Taher, N.; , "Implementing multi-gigahertz test systems using CMOS FPGAs and PECL components," *Solid-State Circuits Conference, 2005. ESSCIRC 2005. Proceedings of the 31st European* , vol., no., pp. 291- 294, 12-16 Sept. 2005

O. Liboiron-Ladouceur; C. Gray; D.C. Keezer; K. Bergman; , "Bit-parallel message exchange and data recovery in optical packet switched interconnection networks," *Photonics Technology Letters, IEEE* , vol.18, no.6, pp.779-781, March 15, 2006

Gray, C.E.; Liboiron-Ladouceur, O.; Keezer, D.C.; Bergman, K.; , "Test electronics for a multi-gbps optical packet switching network," *Electronics Packaging Technology Conference, 2006. EPTC '06. 8th* , vol., no., pp.373-378, 6-8 Dec. 2006

Gray, C.E.; Liboiron-Ladouceur, O.; Keezer, D.C.; Bergman, K.; , "Co-development of test electronics and PCI Express interface for a multi-Gbps optical switching network," *Test Conference, 2007. ITC 2007. IEEE International* , vol., no., pp.1-9, 21-26 Oct. 2007

Keezer, D.C.; Gray, C.; Minier, D.; Ducharme, P.; , "Demonstration of 20 Gbps digital test signal synthesis using SiGe and InP logic," *Mixed-Signals, Sensors, and Systems Test Workshop, 2009. IMS3TW '09. IEEE 15th International* , vol., no., pp.1-5, 10-12 June 2009

Keezer, D.C.; Gray, C.; Majid, A.; Minier, D.; Ducharme, P.; , "A development platform and electronic modules for automated test up to 20 Gbps," *Test Conference, 2009. ITC 2009. International* , vol., no., pp.1-11, 1-6 Nov. 2009

Keezer, David; Gray, Carl; Minier, Dany; Ducharme, Patrice; “Low-Cost 20 Gbps Digital Test Signal Synthesis Using SiGe and InP Logic”, *Journal of Electronic Testing,* vol. 26, issue 1, pp.87-96. Feb. 2010.

Keezer, D.C.; Gray, C.E.; , "Extending a DWDM optical network test system to 10 Gbps ×4," *Mixed-Signals, Sensors and Systems Test Workshop (IMS3TW), 2010 IEEE 16th International* , vol., no., pp.1-5, 7-9 June 2010

Wang, H.; Bergman, K.; Gray, C.; Keezer, D.C.; , "Demonstration of end-to-end bit-parallel memory transactions across the ultra-low latency data vortex optical packet

switch," *Optical Fiber Communication (OFC), collocated National Fiber Optic Engineers Conference, 2010 Conference on (OFC/NFOEC)* , vol., no., pp.1-3, 21-25

Keezer, D.C.; Gray, C.E.; , "Two methods for 24 Gbps test signal synthesis," *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011* , vol., no., pp.1-4, 14-18 March 2011

Keezer, D.C.; Gray, C.E.; , "Extending Low-Cost Test Signal Synthesis to 40 Gbps," *Mixed-Signals, Sensors and Systems Test Workshop (IMS3TW), 2011 IEEE 17th International* , vol., no., pp.64-66, 16-18 May 2011

Gray, Carl; Keezer, David; "Extending a DWDM Optical Network Test System to 12 Gbps x4 Channels", *Journal of Electronic Testing,* vol. 27, issue 3, pp. 351-361, June 2011.

Gray, C.; Keezer, D.C.; Wang, H.; Bergman, K.; , "Burst-Mode Transmission and Data Recovery for Multi-GHz Optical Packet Switching Network Testing," *Test Symposium (ATS), 2011 20th Asian* , vol., no., pp.545-551, 20-23 Nov. 2011